Gustavo Oliveira de Souza

# Desenvolvimento de software para indicação médica utilizando arquitetura limpa

Brasil

31 de março de 2021

Gustavo Oliveira de Souza

# Desenvolvimento de software para indicação médica utilizando arquitetura limpa

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia da Universidade Federal de São Paulo como parte das atividades para obtenção do título de Bacharel em Engenharia de Computação.

Universidade Federal de São Paulo – UNIFESP

Instituto de Ciência e Tecnologia

Trabalho de Conclusão de Curso

Orientador: Otávio Augusto Lazzarini Lemos

Brasil

31 de março de 2021

Gustavo Oliveira de Souza
Desenvolvimento de software para indicação médica utilizando arquitetura limpa/ Gustavo Oliveira de Souza. – Brasil, 31 de março de 2021-
83 p. : il. (algumas color.) ; 30 cm.

Orientador: Otávio Augusto Lazzarini Lemos

Trabalho de Conclusão de Curso –
Universidade Federal de São Paulo – UNIFESP
Instituto de Ciência e Tecnologia
Trabalho de Conclusão de Curso, 31 de março de 2021.

1. Arquitetura limpa. 2. Metodologias ágeis. 3. Requisitos. 4. User story. 5. Princípios SOLID. 6. Gestão de Projetos.
I. Universidade Federal de São Paulo. II. Instituto de Ciência e Tecnologia.

— Dedicatória —

*Espiritualmente, dedico minha vida a meu Salvador, Jesus Cristo.*
*Fisicamente, dedico este trabalho a meus pais, Vandembergue e Elieuza, os maiores patrocinadores desse sonho.*

# Agradecimentos

A Deus por me proporcionar perseverança por todo esse curso.

Sou grato aos meus pais, Vandembergue Souza e Elieuza Souza, por sempre me incentivarem e acreditarem que eu seria capaz de transpor os obstáculos que a vida me apresentou.

Agradeço a Lucas Lellis, Davi Morales, Gabriela Souza, Guilherme Damasceno, Islan Fialho, Cristiane Fialho e a meu orientador Otávio Lemos pelas valiosas contribuições a este trabalho.

Sou grato a meus líderes Matheus Ribeiro, Douglas Ferreira, pastor Carlito Paes e à Igreja da Cidade em São José dos Campos pela mentoria espiritual que foi essencial para que eu conseguisse superar vários desafios em minha vida.

Um agradecimento especial a Lucas Lellis e a Eliseu Gomes, que muito me auxiliaram desde as fundações no conhecimento de computação.

Agradeço ao professor Fábio Cappabianco pela oportunidade de desenvolver a minha primeira iniciação científica em computação, e pelo incentivo em estudar essa área do conhecimento.

Agradeço aos professores da Universidade Federal de São Paulo que estão comprometidos com aa elevada qualidade de ensino, e com ajudar a orientar e motivar os alunos.

Por fim, agradeço a todos os meus amigos do curso de graduação que compartilharam dos inúmeros desafios que enfrentamos, sempre com o espírito colaborativo.

*"Os caminhos da sabedoria são caminhos agradáveis, e todas as suas veredas são paz. A sabedoria é árvore que dá vida a quem a abraça; quem a ela se apega será abençoado."*
*(Bíblia Sagrada, Provérbios 3, 17-18)*

# Resumo

À medida que softwares evoluem, podem adquirir certos tipos de rigidez que podem levar o software ao fim de seu ciclo de vida. A chamada arquitetura limpa é uma das soluções propostas para mitigar ao máximo esse sintoma. O objetivo deste trabalho é construir uma aplicação de indicação médica com essa característica arquitetônica desde sua concepção. O projeto incluiu a pesca de requisitos, gestão de escopo, gestão do cronograma e o acompanhamento da execução, tudo feito de maneira iterativa e ágil. Foi possível analisar que o produto desenvolvido possui sim uma arquitetura limpa, e suas dependências fluem em direção à estabilidade.

**Palavras-chave**: arquitetura limpa. engenharia de software. requisitos. gestão de projetos. *user stories*. metodologias ágeis. princípios *SOLID*.

# Abstract

As a software evolves, it may acquire certain types of rigidity that can bring software to the end of its life cycle. The so-called clean architecture is one of the proposed solutions to mitigate this symptom as much as possible. The goal of this work is to build a medical indication app with this architectural characteristic since its conception. The project includes trawling for requirements, scope management, schedule management and execution monitoring, all obeying agile and iterative principles. It was possible to analyze that the developed product does have a clean architecture, and its dependencies flow towards stability.

**Keywords**: clean architecture. software engineering. requirements. project management. user stories. agile. SOLID principles.

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| ADP | Princípio das Dependências Acíclicas |
| API | Interface de Programação de Aplicações |
| CCP | Princípio do Fechamento Comum |
| CRP | Princípio da Reutilização Comum |
| DIP | Princípio da Inversão de Dependências |
| DRY | *Don't Repeat Yourself* - Não Se Repita |
| EAP | Estrutura Analítica do Projeto |
| GP | Gestor de Projeto |
| IEEE | Instituto de Engenheiros Eletricistas e Eletrônicos |
| ISP | Princípio da Segregação de Interfaces |
| JSON | Notação de Objetos do *JavaScript* |
| KPI | *Key Performance Indicator*, ou Principal Indicador de Performance |
| LSP | Princípio da Substituição de *Liskov* |
| OCP | Princípio Aberto/Fechado |
| PDCA | Ciclo *Plan-Do-Check-Act* |
| REP | Princípio da Equivalência Reutilização/Entrega |
| REST | Transferência Representacional de Estado |
| RPC | Chamada de Procedimento Remoto |
| SAP | Princípio das Abstrações Estáveis |
| SDP | Princípio das Dependências Estáveis |
| SOAP | Protocolo de Acesso a Objetos Simples |
| SRP | Princípio da Responsabilidade Única |
| TDD | Desenvolvimento Guiado a Testes |
| US | *User Story*, ou história de usuário |

# Sumário

# Introdução

A engenharia de software é um tipo de engenharia um tanto peculiar. Enquanto engenharias clássicas consolidam suas técnicas há séculos (FARAH, 2019), a invenção e uso massificado dos computadores é recente (ALVES, 2014).

O advento dos compiladores a partir da década de 1950 (LEMONE, 1992) e das linguagens orientadas a objeto - como C++, a partir da década de 80 - possibilitaram um fenômeno interessante: na engenharia de software, o processo de manufatura da especificação se tornaria extremamente rápido e barato, enquanto outras engenharias seguiriam apresentando custos bem mais elevados (REEVES, 1992).

À medida que a lei de Moore elevava a capacidade computacional a níveis cada vez mais altos (INTEL, 2020) e as tecnologias de projeto de software se simplificavam, a complexidade do softwares também crescia (REEVES, 1992). Hoje há softwares com bilhões de linhas de código (METZ, 2015) sendo utilizados corriqueiramente pelas pessoas.

Essa evolução computacional tornou a mudança de requisitos extremamente comum e isso levou a novos desafios: como projetar soluções de software que resistam às recorrentes mudanças de requisitos? Algumas soluções foram propostas: as metodologias ágeis (MARTIN; MARTIN, 2011); mudanças na documentação de requisitos (COHN, 2004); evolução das arquiteturas de software (MARTIN, 2017); adoção de princípios e padrões de projeto (MARTIN; MARTIN, 2011); utilização de ferramentas modernas de gestão de projetos (INSTITUTE, 2017).

Este trabalho visa a aplicação dessas propostas para a construção de um software de indicação médica.

# 1 Objetivos

## 1.1 Objetivos Gerais

Desenvolver um software de arquitetura limpa com finalidade de facilitar a indicação de médicos a possíveis pacientes, adotando princípios provenientes de metodologias ágeis e gestão de projetos.

## 1.2 Objetivos Específicos

- Explorar uma metodologia ágil para um único desenvolvedor para que ela contemple sessões de trabalho semelhantes a metodologias como *Scrum* e Programação Extrema;
- Investigar a aderência das sessões de planejamento às temáticas de pesca (do inglês, *trawling* (ROBERTSON; ROBERTSON, 2014)), priorização de requisitos e criação de cronograma de lançamento (ou *roadmap*);
- Avaliar o desempenho do desenvolvimento pelas métricas de adequação aos testes de aceitação e gráficos de burndown;
- Avaliar a adequação do software desenvolvido à arquitetura limpa.

# 2 Metodologia

Métodos tradicionais de desenvolvimento como cascata possuem fases bem definidas de desenvolvimento. Essas fases são as seguintes (COHN, 2004):

- Escrita de requisitos;
- Análise de requisitos;
- Desenho da solução;
- Implementação da solução;
- Teste da solução.

Como cada fase do desenvolvimento do software em cascata pode ser bastante demorada, esse tipo de metodologia pode ser avesso ao desejo do cliente de mudar o comportamento do sistema. (COHN, 2004) Alguns tipos de software, como da indústria aeronáutica, podem se beneficiar desse comportamento devido ao seu custoso processo de certificação (GLAS; ZIEMER, 2009).

Os signatários do Manifesto Ágil propuseram em 2001 princípios que resultaram na criação dos métodos ágeis (MARTIN; MARTIN, 2011) como *Scrum* (SUTHERLAND, 2016) e Programação Extrema (BECK, 2004). Embora cada metodologia ágil tenha seus próprios processos, eles compartilham de características comuns, como desenvolvimento incremental das soluções, permitindo uma rápida mudança de comportamentos do sistema no ciclo de desenvolvimento(COHN, 2004).

Para a realização deste trabalho, propõe-se uma metodologia semelhante à Programação Extrema e Scrum, com alguns refinamentos inerentes à aplicação a um único desenvolvedor.

## 2.1 Planejamento Inicial

Para ser ágil, é preciso ter um relacionamento com o cliente que permita ao desenvolvedor descobrir e entregar valor ao longo do processo de desenvolvimento. Para que isso ocorra, o primeiro passo é o estabelecimento de um time de clientes (COHN, 2004).

Propõe-se como meta para este trabalho um time de clientes composto por 2 médicos e 2 pessoas que não trabalham na área da saúde.

Uma vez que o time do cliente esteja estabelecido, a primeira atividade é uma reunião de exploração das funcionalidades. Essa exploração inicial será feita numa sessão de trabalho com o desenvolvedor e o time de desenvolvimento que possuirá os seguintes momentos:

- Escrita de histórias de usuário (ou *user stories*) com nível extremamente baixo de detalhes;
- Organizar *user stories* e ordenar as prioridades conforme avaliação do time de clientes;
- Estabelecer estimativas de prazos e complexidade para cada história, criando um *roadmap*.

## 2.2 Iterações

Em processos ágeis, o processo de entrega de valor é feito em iterações. Cada iteração tem um tempo pré-determinado de duração. No método Scrum, a iteração é chamada de *Sprint* (COHN, 2004). Estabeleceu-se a duração da *sprint* em duas semanas.

Para iniciar a iteração, acontece uma **reunião de planejamento da iteração**, que deve:

- Selecionar as *user stories* a serem desenvolvidas. A quantidade de histórias depende do indicador de velocidade de desenvolvimento (dada pela unidade de complexidade por iteração);
- Se necessário, quebrar a história em outras menores;
- Quebra das histórias em tarefas e conferir estimativas usando comparação de medidas;
- Estabelecimento dos testes de aceitação.

Ao longo da iteração, espera-se que haja dúvidas sobre a funcionalidade e que detalhes de cada história sejam discutidos com o cliente. Espera-se também que a evolução dos esforços seja contabilizada.

Após a *sprint*, promove-se uma **reunião de finalização da iteração**, que deve:

- Apresentar a funcionalidade nova que esteja funcionando 100%;
- Avaliar os testes de aceitação;
- Analisar a performance da iteração e recálculo do indicador de velocidade;

- Avaliar o roadmap e quebrar os épicos (histórias de usuários que podem ser subdividas em outras) conforme eles se aproximam da execução, aumentando o nível de detalhe para as próximas iterações.

## 2.3 Consolidação

Ao final de uma data limite, o projeto passará por uma fase de finalização e avaliação do software, onde será avaliada a aderência à Arquitetura Limpa (MARTIN, 2017), auxiliada por métricas relativas à análise de dependência entre classes e componentes.

# 3 Fundamentação Teórica

## 3.1 Requisitos

O primeiro passo para a construção de um software é a definição do que ele pode fazer, sem a qual um desenvolvedor não saberá o que codificar. Para que esse software seja útil, deverá possuir certos comportamentos e qualidades desejáveis a seus usuários. A essas características, pode-se dar o nome de requisitos (ROBERTSON; ROBERTSON, 2014).

Para Robertson & Robertson (ROBERTSON; ROBERTSON, 2014), existem três tipos de requisitos:

- Requisitos Funcionais;
- Requisitos Não-Funcionais;
- Restrições (ou *Constraints*, em inglês).

Requisitos funcionais dizem respeito ao que o programa é capaz de fazer (por exemplo, permitir ao usuário comprar produtos é um requisito funcional importante para um sistema de compras). Os não-funcionais se referem aos objetivos de qualidade (metas de performance podem ser requisitos não-funcionais). Restrições, por sua vez, são assuntos sobre o projeto em si que são capazes de influenciar o projeto do software (um exemplo de restrição para um site poderia ser a capacidade de rodar no celular e no computador).

Há diversas maneiras de se comunicar requisitos entre as partes interessadas (ou *stakeholders*, em inglês) no desenvolvimento de um software. Três maneiras notáveis de promover esse tipo de comunicação são: histórias de usuário (ou *user stories*, em inglês) (COHN, 2004), casos de uso (ou *use cases*, em inglês) (COCKBURN, 2000) e a padronização IEEE 830 (IEEE, 1998).

### 3.1.1 *User Stories*

A principal característica de uma *user story* (em português, história de usuário) é de ser um requisito com poucos detalhes de funcionamento. Como consequência, metodologias de desenvolvimento que adotem esse método acabam tendo que se comunicar bastante com o cliente (COHN, 2004). Esse efeito é cobiçado por metodologias ágeis como a Programação Extrema (MARTIN; MARTIN, 2011).

Cohn (COHN, 2004) descreve os três componentes de uma história de usuário:

- Breve descrição escrita da história usada no planejamento e lembretes, normalmente escrita em cartões;
- Conversas que servem para revelar detalhes da história;
- Testes de aceitação, que documentam detalhes e são usados para determinar a completude de uma história.

A Figura 1 exemplifica uma *user story* e seus componentes escritos. Nela, nota-se que um dos testes de aceitação foi especialmente escrito devido a um lembrete, que teve como consequência uma conversa com o cliente desse software.

| Um cliente pode pagar por uma compra com cartão de crédito | |
|---|---|
| Nota<br>Nota para UI | Aceitaremos Elo?<br>Descobrir a bandeira do cartão pelos primeiros dígitos |
| Testes de Aceitação | • Tentar pagar com Visa e Master (passar)<br>• Tentar pagar com Elo (falhar)<br>• Tentar pagar com cartões de números válidos e inválidos (ruim ou incompleto)<br>• Tentar pagar com cartões vencidos (falhar)<br>• Tentar pagar compras de valores acima do limite do cartão (falhar) |

Figura 1 – Exemplo de história de usuário para um sistema de compras

Adaptado de (COHN, 2004).

#### 3.1.1.1 Boas Práticas na escrita de *user stories*

Wake (WAKE, 2003) elencou cinco critérios importantes para se ter em mente ao se escrever *user stories*, além de ter criado o acrônimo mnemônico **INVEST**, que revela esses critérios. Ao lembrá-lo, tem-se o significado que boas histórias devem ser:

- **I**ndependentes

- **N**egociáveis
- **V**aliosas para usuários ou clientes
- **E**stimáveis
- **S**ucintas
- **T**estáveis

Sendo independentes, histórias não se sobrepõem, portanto podem ser desenvolvidas em qualquer ordem. Sendo negociáveis, histórias podem mudar facilmente. Assim sendo, não documentam um contrato de desenvolvimento.

Sendo valiosas para clientes, histórias não focam em gerar valor para o desenvolvedor. Nem mesmo quando significa mexer em partes diferentes da *stack* (em sistemas web, dá-se esse nome para representar que o sistema trabalha com uma pilha de tecnologias, que incluem aplicações *front-end*, *back-end* e o banco de dados (DEVMEDIA, 2020) ).

Sendo estimáveis, histórias são desenvolvidas com alguma ideia de tempo de desenvolvimento. A falta de conhecimento sobre algum domínio pode inviabilizar a estimativa. Esse obstáculo pode ser transposto com o uso de histórias investigativas (ou *spike story*, em inglês).

Sendo sucintas, histórias apresentam clareza e adiam complexidades desnecessárias. Quando são amplas, são essas histórias passam a ser denominadas épicos (COHN, 2004), que devem ser subdivididos em outras histórias. Sendo testáveis, especialmente com o uso de testes automatizados (COHN, 2004), histórias são facilmente verificáveis.

Wake (WAKE, 2003) defende que histórias devem proporcionar histórias fim-a-fim (ou *end-to-end*, em inglês). Ele usa a metáfora que diz que uma história seria como uma fatia de bolo recheado, onde cada camada do bolo seria como uma aplicação da *stack*. Lauesen (LAUESEN, 2002), por sua vez, defende que histórias devem ser completas, ou seja, devem proporcionar ao usuário um sentimento de realização de um objetivo que antes não seria possível.

Newkirk e Martin (NEWKIRK; MARTIN, 2001) defendem o uso de histórias também como requisitos de restrição, que não precisam de implementação para serem estabelecidas. Para eles, esses cartões (de histórias) servirão como lembrete a serem úteis durante a elaboração de testes de aceitação. Um exemplo de história de restrição é: "O novo sistema deve reaproveitar a base de dados atual".

Cohn (COHN, 2004) defende que uma *user story* deve ser escrita pelo cliente, com um papel de usuário (ou *user role*, em inglês, que é uma coleção de atributos que caractecterizam uma população de usuários e suas intenções de interação com o sistema) explícito, sempre no singular (evitando grupos de usuários) e com voz ativa (evitando voz

passiva). Também ilustra que a sintaxe de Connextra, que pode ser estereotipada pela forma (ALLIANCE, 2001) "Como (papel de usuário), desejo (funcionalidade) para que (valor do negócio)", obedece a essas recomendações.

O mesmo autor ainda sugere que a amplitude do escopo deve ser proporcional ao tempo para iniciar o desenvolvimento. Ou seja, quanto mais próximo o desenvolvimento, menor (e mais específica) a história.

### 3.1.2 Casos de Uso

Casos de uso são uma forma de se documentar requisitos introduzida em 1992 por Jacobson (JACOBSON, 1992). São caracterizados pela escrita num formato aceitável por clientes e desenvolvedores. Descrevem de maneira generalizada um conjunto de interações entre o sistema e seus atores, que diferentemente dos *user roles* podem ser outros sistemas ou usuários (COHN, 2004).

A Figura 2 mostra um exemplo de caso de uso. Nela, destaca-se a presença dos campos: ator primário, escopo, nível, *stakeholders* e interesses, precondições, garantias mínimas, garantias de sucesso, curso primário e cursos alternativos (ou extensões).

Para Martin e Martin (MARTIN; MARTIN, 2011), as duas partes mais importantes de um caso de uso são, respectivamente, o curso principal e os cursos alternativos do sistema. O primeiro diz respeito à sequência visível de eventos sem que nada dê errado para alcançar um objetivo. O segundo, por sua vez, revela detalhes acerca de fatores que poderiam falhar. Os autores ainda defendem que essas duas partes são suficientes para o desenvolvimento, e que os demais campos são sofisticações que são melhor aplicadas por desenvolvedores com mais experiência na utilização de casos de uso.

Para Cohn (COHN, 2004), à medida que histórias de usuário amadurecem e ganham seus testes de aceitação, torna-se natural uma evolução de histórias de usuário para um caso de uso essencial, que o autor define como um tipo de caso de uso desprovido de suposições tecnológicas e detalhes de implementação. Afinal, para (COCKBURN, 2000), casos de uso funcionam como contrato que documentam as funcionalidades acordadas, diferentemente de *user stories*.

### 3.1.3 IEEE 830

A notação de requisitos pela padronização IEEE 830 é marcada por descrições formais dos atributos de um produto. Fazem o constante uso da sintaxe "O sistema deve ..."(COHN, 2004). A Figura 3 mostra um exemplo de requisitos nesse padrão.

Cohn (COHN, 2004) caracteriza esse tipo de comunicação de requisitos como hostil à mudança de requisitos, tão comuns em desenvolvimento de software. Carroll (CARROLL, 1994) ainda aponta a dificuldade de compreensão do sistema pois cada requisito poderia

Figura 2 – Exemplo de caso de uso para compra de ações na internet

Ator Primário: Comprador

Escopo: Conselheiros pessoais/ Pacote Financeiro ("PAF")

Nível: Objetivo do usuário

Partes interessadas e interesses:

Comprador – Quer comprar ações, adicione-os ao portfólio PAF automaticamente

Corretora de Ações – Quer informações completas da operação

Pré-condição: Usuário já tem PAF aberto

Garantia mínima: Registro de informações suficientes para que o PAF consiga detectar possíveis erros e pedir ao usuário por detalhes

Garantia de sucesso: Site remoto reconhece a compra, os registros e o portfólio do usuário são atualizados

Cenário de sucesso principal:

1. Usuário seleciona para comprar ações na web
2. PAF retira a informação do nome do site a se usar do usuário
3. PAF abre conexão ao site, retendo controle
4. Usuário navega e compra ações do site
5. PAF intercepta respostas do site e atualiza o portfólio do usuário
6. PAF mostra o novo portfolio do usuário

Extensões

2a. Usuário solicita ao PAF um site não suportado

2a1. Sistema tira nova sugestão do usuário, com opção de cancelar o caso de uso

3a. Falha web durante configuração:

3a1 Sistema reporta falha ao usuário, volta um passo

3a2 Usuário sai do caso de uso ou tenta novamente

4a. Computador quebra ou é desligado durante transação

4a1 (o que fazer aqui?)

4b Site não reconhece compra, mas deixa em espera

4b1 PAF registra o atraso, começa a contagem no cronômetro para perguntar ao usuário sobre resultados

4b2 Ver caso de uso Atualizar pergunta de compra

5a Site não retorna informações necessárias da compra

5af PAF registra a falta de informações, pede para o usuário Atualizar pergunta de compra

Fonte: (COCKBURN, 2000)

levar à imaginação de um sistema diferente do que deveria ser. Essas características tornam a IEEE 830 não muito adequadas para metodologias ágeis de desenvolvimento (COHN, 2004).

Figura 3 – Exemplo de requisitos no padrão IEEE 830

4.6) O sistema deve permitir a uma empresa pagar por um anúncio de emprego com cartão de crédito

4.6.1) O sistema deve aceitar os cartões Visa, MasterCard e American Express

4.6.2) O sistema deve cobrar do cartão antes de publicar o anúncio

4.6.3) O sistema deve fornecer um número de confirmação único

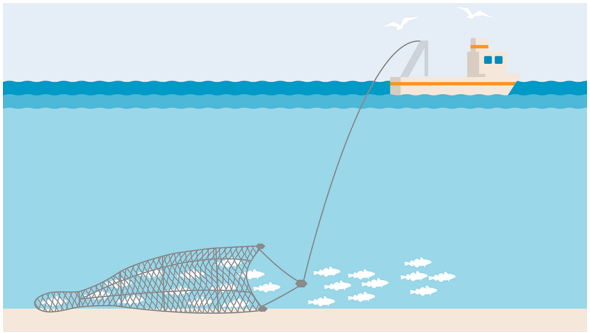
Fonte: (COHN, 2004)

### 3.1.4 Pesca de Requisitos

#### 3.1.4.1 Nomenclatura

Uma das principais atividades no ramo do desenvolvimento de software é a que define os requisitos para o software. Essa atividade possui algumas nomenclaturas distintas: elicitação (KOVITZ, 1998), captura (JACOBSON; BOOCH; RUMBAUGH, 1999) ou ainda pesca (ROBERTSON; ROBERTSON, 1999) (do inglês, *trawling*, que significa pesca com rede em um barco como ilustrado na Figura 4) de requisitos.

Figura 4 – *Trawling*, ou pesca com rede em um barco



Fonte: (COUNCIL, 2020)

Cohn (COHN, 2004) condena a utilização dos termos elicitação e captura enfaticamente, em suas palavras "Elicitation and Capture Should Be Illicit"(em tradução livre, "Elicitação e Captura devem ser Ilícitos"). Também defende a nomenclatura de pesca apontando que a metáfora traz boas consequências:

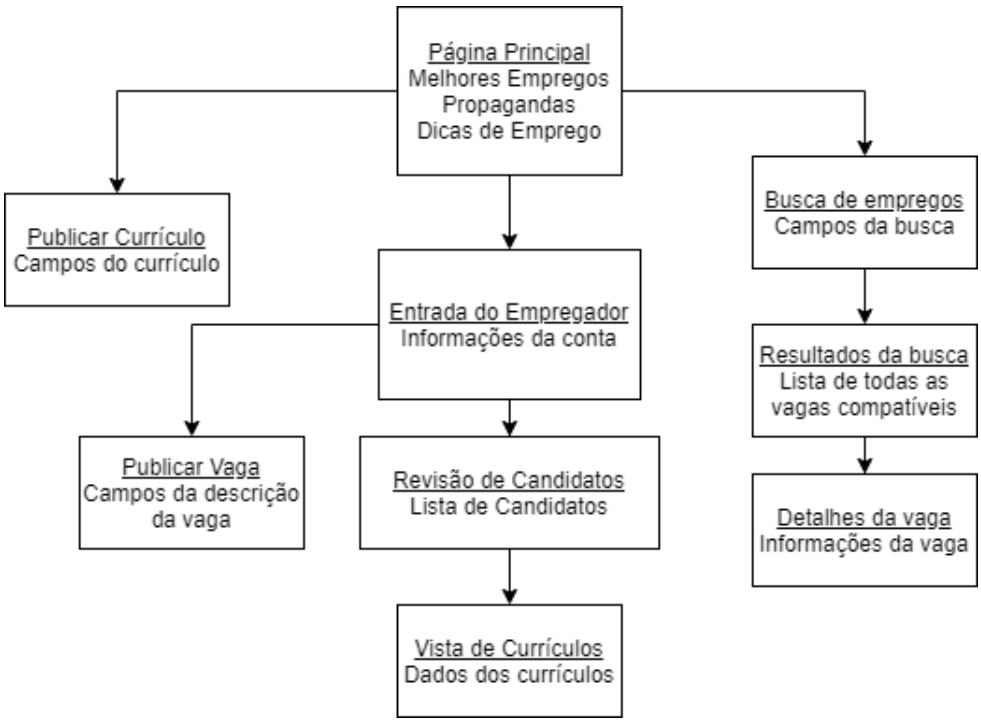
- Quantidade é essencial para o processo de pesca, mas é impossível pescar tudo;

- Requisitos, assim como os peixes, amadurecem e podem morrer;
- Bons pescadores sabem onde achar requisitos enquanto maus pescadores perderão muito tempo.

#### 3.1.4.2 Técnicas

O processo de pesca de requisitos pode ser visto como um tipo de prospecção de negócio. Analistas de empresas possuem um ferramental capaz de auxiliar nesse processo, como o uso de entrevistas, questionários, observação e *workshops*, Embora todas apresentem suas vantagens e desvantagens, a última se destaca por permitir uma participação mais ativa do cliente através de processos do que o autor denomina como prototipação de baixa fidelidade, como ilustrado na Figura 5 (COHN, 2004).

Figura 5 – Exemplo de protótipo de baixa fidelidade




Fonte: (COHN, 2004)

### 3.1.5 Prototipação

Em processos ágeis de desenvolvimento de software, o gerenciamento de requisitos ganha uma dinâmica de obtenção e validação de requisitos cíclicos (MANIFESTO, 2001). Para a comunicação de ideias com o cliente, alguma representação do sistema poderia ser necessária em qualquer parte do ciclo.

Walker, Takayama e Lames (WALKER; TAKAYAMA; LANDAY, 2002) definem o termo protótipo como um modelo de trabalho utilizado para desenvolver e testar ideias de

*design*. Diferentes técnicas de prototipação são utilizadas por desenvolvedores de *software* ao longo do desenvolvimento de produto (NEWMAN; LANDAY, 2000).

À medida que o protótipo evolui, ele se torna mais difícil de ser distinguido do produto final. Essa dificuldade de distinção é o que (WALKER; TAKAYAMA; LANDAY, 2002) definem como a **fidelidade** de um protótipo. Ou seja, enquanto um protótipo de baixa fidelidade pode ser facilmente distinguido do produto derradeiro, outro de alta fidelidade apresentaria alguma dificuldade em passar pelo mesmo processo de discernimento.

(NEWMAN; LANDAY, 2000) alertam que o termo protótipo poderia ser considerado ambíguo. Essa ambiguidade decorre dele poder significar tanto "qualquer coisa que represente o sistema", quanto os chamados *protótipos interativos*, que permitem ao *designer* avaliar como o usuário irá interagir com o produto. Os autores ainda defendem que o segundo significado é mais utilizado.

Apesar de (BABICH, 2017) definir protótipo como uma expressão de intenção de *design*, o autor parece concordar com a desambiguação proposta por (NEWMAN; LANDAY, 2000). A consequência da concordância é que aquele autor defende que ativos estáticos não podem ser considerados protótipos.

A técnica de prototipação estática de baixa fidelidade apresentada por (COHN, 2004) e ilustrada na Figura 5 se assemelha ao que (NEWMAN; LANDAY, 2000) define como *site map*. Para este autor, esta técnica é utilizada no intuito de refletir o entendimento da estrutura de informação em um site.

## 3.2 Planejamento

Quando se interpreta que o processo de desenvolvimento de software se trata de um esforço realizado em um período delimitado de tempo para a criação de um produto, tem-se como consequência que se está se executando um projeto (INSTITUTE, 2017).

Existem técnicas de gerenciamento de projetos bem consolidadas que se tornam úteis para desenvolvimento de software. Um bom gerente de projetos deve ser capaz de gerenciar (INSTITUTE, 2017):

- Integração;
- Escopo;
- Cronograma;
- Custos;
- Qualidade;

- Comunicações;
- Riscos;
- Aquisições;
- Partes interessadas (ou *stakeholders*, em inglês).

### 3.2.1 Gestão de escopo com *user stories*

Escopo, em gestão de projetos, está intimiamente ligado a requisitos. A limitação de tempo força a escolha de quais requisitos serão desenvolvidos em determinado tempo. Um método para realizar esse gerenciamento é da criação de um documento denominado Estrutura Analítica do Projeto (ou EAP). Esse documento é responsável por agrupar funcionalidades em blocos denominados **entregáveis** (INSTITUTE, 2017).

Cohn (COHN, 2004) defende que os entregáveis são constituídos de épicos (histórias de usuário amplas) ou temas (BECK, 2004) (áreas de foco). Como metodologias ágeis se esforçam para entregar valor incrementalmente, priorizando as atividades que mais agregam valor (MARTIN; MARTIN, 2011); a gestão de escopo com *user stories* é suportada pela organização e prioridade das funcionalidades.

A metodologia ágill DSDM (CONSORTIUM; STAPLETON, 2003) adota um sistema de priorização que pode ser útil na gerência de escopo denominado **MoSCoW** (semelhante à palavra Moscou, capital da Rússia, em inglês). Nesse sistema, cada épico deve receber para um entregável um dos quatro rótulos definidos pelo acrônimo:

- ***M****ust* - Obrigação
- ***S****hould* - Altamente provável
- ***C****ould* - Probabilidade média ou baixa
- ***W****on't* - Não é uma prioridade para o entregável

### 3.2.2 Gerenciamento de cronograma com *user stories*

A frase "Não se gerencia o que não se mede, não se mede o que não se define, não se define o que não se entende, e não há sucesso no que não se gerencia" atribuída a William Edwards Demming é extremamente relevante em gestão, especialmente gestão de tempo. Portanto, a única forma de se gerenciar o cronograma é através de medidas de tempo.

Vimos que USs devem seguir os critérios INVEST (WAKE, 2003). Veremos a seguir como a independência e estimabilidade são características que tornam a adoção desse tipo de requisito útil para a gestão de projetos.

Se estima uma história pensando na quantidade de tempo necessária para realizá-la. O valor pode ser em medidas completamente subjetivas, ou ser proporcional a dias de trabalho, mas é importante que o critério de avaliação seja uniforme para a equipe de desenvolvimento durante toda a execução do projeto. Uma técnica proposta por Cohn (COHN, 2004) para garantir a uniformidade do critério é a adoção de comparações (que o autor denomina triangulações) entre medidas de histórias associado com o uso de valores pré-estabelecidos em alguma sequência pré-acordada (como Fibonacci).

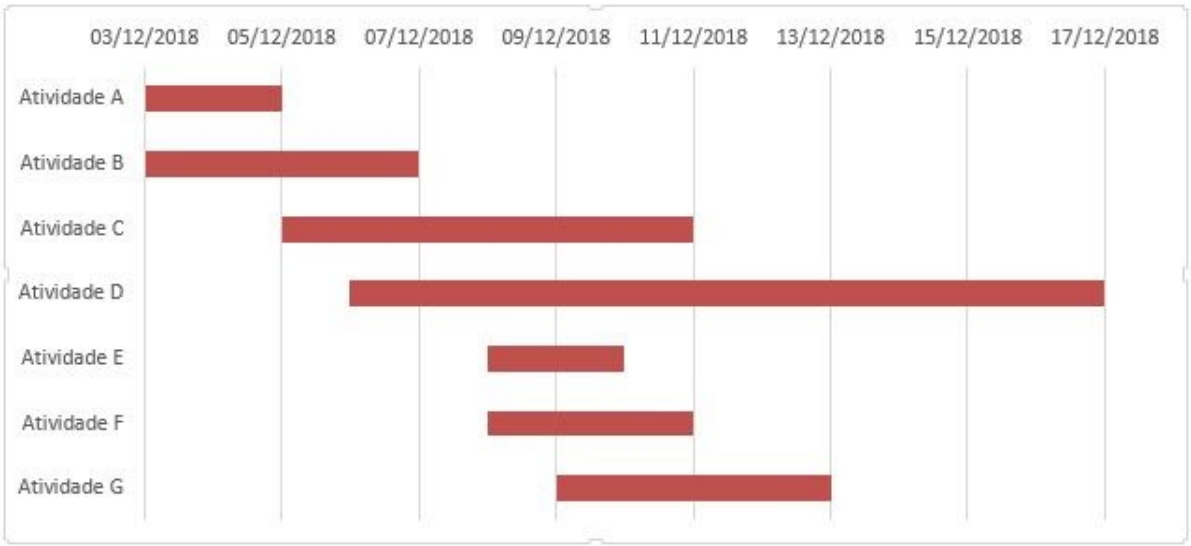
Pode ser importante recalcular estimativas caso se opte por mudar a ordem de execução de USs (por exemplo, o tempo para se lavar roupa muda se uma máquina de lavar for instalada antes ou depois da lavagem) (COHN, 2004).

Tendo estimativas, é possível estabelecer prazos para o lançamento de entregáveis em datas denominados *milestones* e de algumas métricas que podem ser consideradas KPIs (do inglês, *Key Performance Indicators*, ou Principais Indicadores de Performance) (INSTITUTE, 2017).

Metodologias ágeis possuem ainda um outro nível de entrega, que é o de iteração (que para a maioria das metodologias é um prazo fixo entre 2 e 3 semanas). Disso, pode-se calcular o KPI de velocidade de execução medindo-se a complexidade entregue no prazo (usando-se a estimativa inicial) em cada iteração (MARTIN; MARTIN, 2011).

A velocidade de execução do projeto dita quantas USs um time de desnvolvimento se comprometem a fazer numa iteração, o prazo (em iterações) para os *milestones* e até mesmo o prazo para o fim do projeto. Variações nessa velocidade são interpretadas pelo GP (Gestor de Projeto) como risco. Esse gerente pode tomar atitudes como paralelizar atividades utilizando um diagrama de Gantt, ilustrado na Figura 6 (INSTITUTE, 2017).

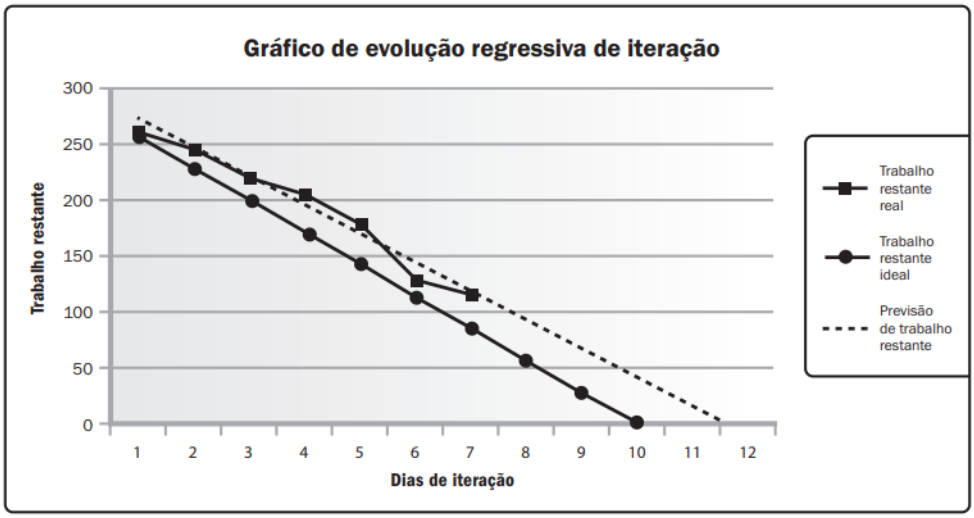
Figura 6 – Exemplo de Diagrama de Gantt




Fonte: (LIMA, 2018)

Gerentes podem averiguar a mudança nas velocidades durante uma iteração através de um gráfico de evolução regressiva (burndown) de iteração, ilustrado na Figura 7. Um gráfico semelhante pode ser usado para medir o progresso do projeto, no gráfico de *burndown* do projeto.

Figura 7 – Exemplo de gráfico de evolução regressiva, ou gráfico de *burndown*




Fonte: (INSTITUTE, 2017)

### 3.2.3 Gestão da Qualidade

Em uma organização, o sistema de gestão de qualidade reúne processos, procedimentos e hierarquias de modo a coordenar as atividades da organização. Essa coordenação tem o intuito de levá-la a ser mais efetiva e eficiente em satisfazer objetivos de satisfação do cliente, obediência a regulações, etc (ASQ, 2021).

Dentro do contexto industrial, surgiu uma filosofia japonesa denominada *Kaizen* (que em japonês significa "mudança para melhor"). Essa filosofia prega a diminuição dos desperdícios e melhoria contínua dos processos envolvendo a linha de produção (GONÇALVES, 2017).

Um dos métodos empregados pela filosofia *Kaizen* é o ciclo *PDCA*. Essa sigla é um acrônimo para *Plan-Do-Check-Act*. Esse método é utilizado para resolver problemas em 4 etapas (ENDEAVOR, 2020), listadas abaixo. Essas etapas também estão ilustradas na Figura 8.

- Planejamento, onde o processo de melhoria é planejado
- *Do* (do inglês, fazer) onde a ação é posta em prática
- Checagem, onde a ação planejada é medida

- Ajuste, onde se usa as medidas tomadas no passo anterior para fazer ajustes

Figura 8 – Infográfico do ciclo do *PDCA*


Fonte: (INSIDER, 2020)

## 3.3 Engenharia de *Software*

### 3.3.1 Design de *software*

O processo de construção de *software* não é tão diferente de um processo de engenharia. Reeves (REEVES, 1992) afirma que o fruto do trabalho de um engenheiro é a especificação de uma solução; em engenharia de software essa especificação não são comentários no código, mas o código fonte em si. O autor ainda afirma que na engenharia, a especificação da solução é enviada para o processo de manufatura; para a engenharia de software, o compilador (ou interpretador, para linguagens interpretadas) é a manufatura, e o produto final é o código executável.

A facilidade de se construir software fez com que os projetos de *software* ganhassem complexidade. Softwares usados corriqueiramente como Google e Windows possuem pelo menos, respectivamente, 2 bilhões e 45 milhões de linhas de código (METZ, 2015).

#### 3.3.1.1 Maus cheiros do projeto

Essa enorme complexidade atrelada a requisitos constantemente mudando fazem, ao longo do tempo, o software se tornar mais difícil de se manter e ganhar novas funcionalidades. Martin e Martin (MARTIN; MARTIN, 2011) alegam que softwares que alcançam esse estágio possuem os seguintes sintomas:

- Rigidez : característica de um software onde uma alteração em um módulo provoca alterações em outros.
- Fragilidade : Um software é considerado frágil quando uma alteração em um lugar provoca erro em outros lugares.
- Imobilidade : Um software é imóvel quando uma parte útil não pode ser reaproveitada em outro lugar por ser difícil de isolá-la.
- Viscosidade : acontece quando soluções malfeitas tem implementação facilitada em relação a soluções que preservam o projeto. Viscosidade de ambiente ocorre quando o ambiente de configuração do projeto promove perdas de produtividade.
- Complexidade Desnecessária : Um software apresenta complexidade desnecessária quando possui elementos não utilizados.
- Repetição desnecessária : Um software apresenta repetição desnecessária quando possui o mesmo código replicado em vários lugares, com poucas modificações, o que é um indicativo de ausência de abstrações.
- Opacidade : É dito que o módulo de um software é opaco quando é difícil de ser compreendido.

### 3.3.2 Princípios *SOLID*

Martin (MARTIN, 2017) organizou cinco princípios que dão ao software uma maior tolerância às mudanças em um único acrônimo SOLID, que se refere a:

- **S**ingle Responsability Principle (SRP)- Princípio da Responsabilidade Única : cada classe só deve fazer uma coisa, portando deverá possuir apenas um único motivo para ser alterada (MARTIN; MARTIN, 2011).

- **O**pen/Closed Principle (OCP) - Princípio Aberto/Fechado : módulos de um sistema (classes, funções, etc..) devem ser fechados para modificação de comportamentos, mas abertos para extensão/ampliação de funcionalidades. Esse princípio não pode ser 100% cumprido pois as mudanças podem ser ineseperadas (MARTIN; MARTIN, 2011).

- **L**iskov's Substitution Principle (LSP)- Princípio da Substituição de Liskov : Classes filhas são intercambiáveis com seus pais, tanto em contexto de herança quanto de implementação de interfaces. Nas palavras de Liskov (LISKOV, 1987): "O que se deseja aqui é algo como a seguinte propriedade de substituição: se para cada objeto o1 do tipo S existe um objeto o2 do tipo T, tal que, para todos os programas P definidos em termos de T, o comportamento de P fica inalterado quando o1 é substituído por o2, então S é subtipo de T".

- **I**nterface Segregation Principle (ISP) - Princípio da Segregação de Interfaces : classes não devem implementar interfaces que possuem métodos que elas não precisam implementar. Quando esse for o caso, vale a pena dividir as interfaces e as classes implementarão todas as interfaces necessárias (MARTIN; MARTIN, 2011).

- **D**ependency Inversion Principle (DIP) - Princípio da Inversão de Dependência : políticas de alto nível (módulos ou abstrações) não devem depender de detalhes de implementação. Ao contrário, o detalhe depende da política (MARTIN; MARTIN, 2011).

### 3.3.3 *Design* de Componentes

Conforme a complexidade do sistema aumenta, o número de classes e de pessoas desenvolvendo o sistema também aumenta. Uma solução para essa complexidade é agrupar classes em pacotes denominados componentes. Esse tipo de pacote possibilita o desenvolvimento paralelo de partes diferentes do sistema e comumente podem ser implantados individualmente através de DLLs, JARs, etc (MARTIN; MARTIN, 2011).

Como os componentes subdividem o sistema, é comum se adotar o termo granularidade se referindo ao tamanho de um componente: componentes pequenos possuem granularidade fina ou pequena, enquanto grandes componentes possuem granularidade grossa ou grande.

#### 3.3.3.1 Coesão

Ao se criar componentes, é estabelecido uma fronteira (ou *boundary*, em inglês) que delimita quais classes estarão dentro do componente e quais estarão fora. Martin e Martin (MARTIN; MARTIN, 2011) definem 3 princípios para escolher quais classes incluir, tendo como objetivo a coesão do componente.

##### 3.3.3.1.1 Princípio da Equivalência Reutilização/Entrega (REP)

O tamanho (ou granularidade) do reuso de um componente deve equivaler ao tamanho da entrega. Em termos práticos, componentes precisam possuir rótulos de versão para que seus utilizadores saibam o que esperar ao se utilizar desse pacote (MARTIN, 2017).

Para critérios de separação de classes, o REP (Princípio da Equivalência Reutilização/Entrega, do inglês *Release-Reuse Equivalence Principle*) quais componentes devem ser reutilizáveis ou não. Para o componente ser reusável, todas suas classes devem possuir essa mesma característica (MARTIN; MARTIN, 2011).

##### 3.3.3.1.2 Princípio da Reutilização Comum (CRP)

O CRP (Princípio da Reutilização Comum, do inglês, *Common Reuse Principle*) diz que em um componente coeso todas suas classes são reusadas juntas. Portanto, deve-se retirar do componente classes que não possuem relacionamentos fortes com as outras classes do componente. Outra consequência é que deve-se incluir no componente classes que tem relacionamentos fortes entre si (MARTIN; MARTIN, 2011).

##### 3.3.3.1.3 Princípio do Fechamento Comum (CCP)

Semelhante ao SRP para componentes, o CCP (Princípio do Fechamento Comum, do inglês, *Common Closure Principle*) afirma que em um componente coeso, suas classes devem estar fechadas para os mesmos tipos de mudanças. O objetivo desse princípio é evitar que uma mudança em um componente exija mudanças em outros (MARTIN, 2017).

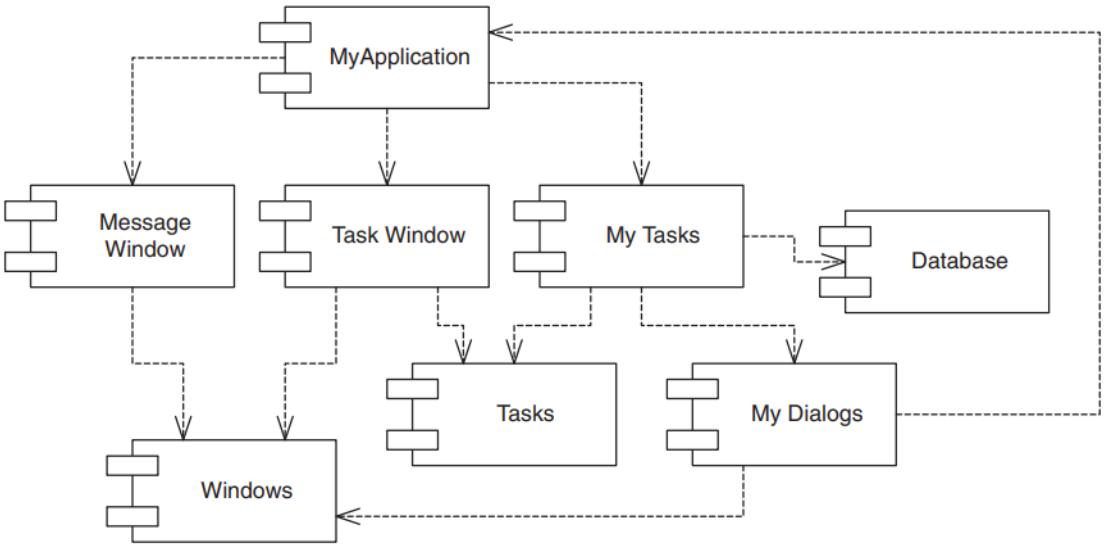
#### 3.3.3.2 Acoplamento

Para buscar uma boa estabilidade em um sistema mesmo com a mudança dos requisitos, é necessário que seus componentes interajam bem entre si. Como a interação entre componentes é feito por formas de acoplamento, Martin e Martin (MARTIN; MARTIN, 2011) definiram 3 princípios que buscam a estabilidade, que veremos a seguir.

##### 3.3.3.2.1 Princípio das Dependências Acíclicas (ADP)

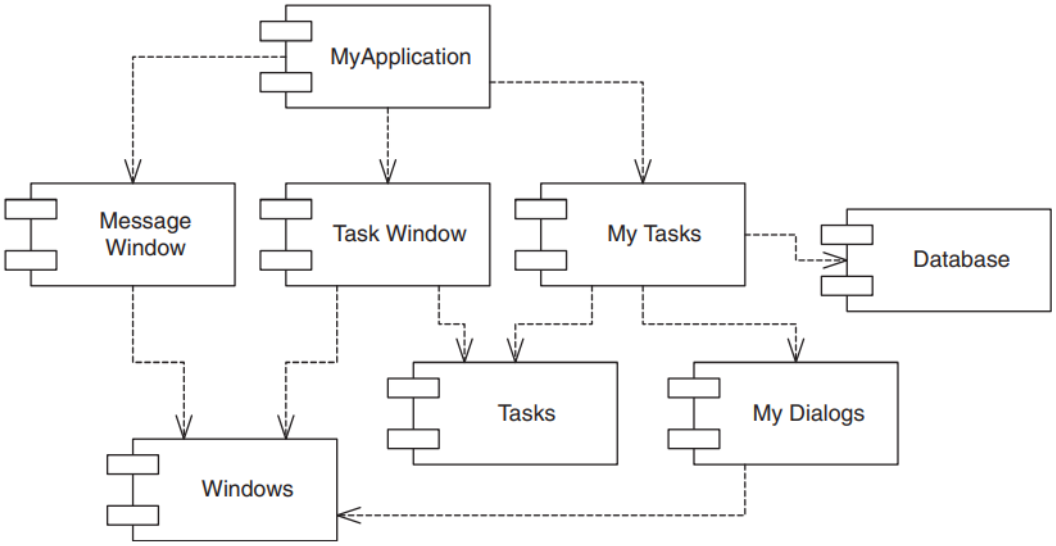
O ADP (Pincípio das Dependências Acíclicas, do inglês, *Acyclic Dependencies Principle*) diz que as dependências entre os componentes de um sistema, que são uma espécie de grafo direcionado, não devem formar ciclos. A Figura 9 ilustra um sistema que não contempla o ADP por possuir ciclos de dependências, especialmente no pacote "My Dialogs", que fecha o ciclo dependendo de "My Application". A Figura 10 ilustra um

Figura 9 – Estrutura de componentes com ciclo de dependências




Fonte: (MARTIN; MARTIN, 2011)

Figura 10 – Estrutura de componentes sem ciclo de dependências



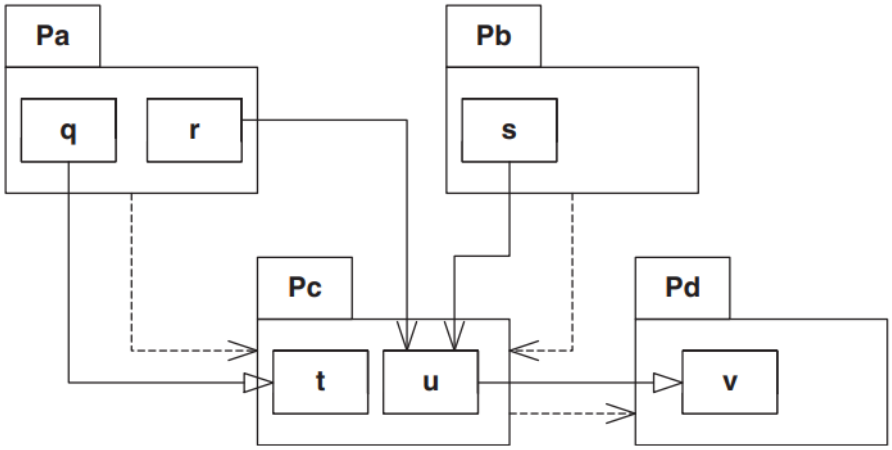

Fonte: (MARTIN; MARTIN, 2011)

sistema semelhante que atende ao princípio por não possuir ciclos (MARTIN; MARTIN, 2011).

#### 3.3.3.2.2 Princípio das Dependências Estáveis (SDP)

O SDP (Princípio das Dependências Estáveis, do inglês, *Stable Dependencies Principle*) dita que as dependências entre componentes de um sistema devem ir em direção

Figura 11 – Diagrama de pacotes útil para o cálculo de instabilidade




Fonte: (MARTIN; MARTIN, 2011)

à estabilidade. Um componente. Quando um componente é alvo de muitas dependências, sua estabilidade aumenta. Quando o componente depende de muitos outros, sua estabilidade diminui (MARTIN; MARTIN, 2011). Na Figura 10 podemos ver que o componente mais instável é o "MyApplication"e os mais estáveis são os componentes "Windows", "Tasks"e "Database".

#### 3.3.3.2.3 Princípio das Abstrações Estáveis (SAP)

O SAP (Princípio das Abstrações Estáveis, do inglês, *Stable Abstractions Principle*) diz que cada componente de um sistema deve possuir um nível de abstração (medido pela quantidade de classes abstratas ou interfaces dentro do componente) equivalente ao seu nível de estabilidade (MARTIN; MARTIN, 2011).

### 3.3.3.3 Métricas

Martin e Martin(MARTIN; MARTIN, 2011) definem as instabilidade e abstração como métricas primárias do *design de componentes*; também define distância da sequência principal e a distância normalizada como métricas secundárias (MARTIN; MARTIN, 2011).

#### 3.3.3.3.1 Instabilidade

Instabilidade mede quanto um componente precisa mudar caso haja alteração em outros componentes. Para medi-la, é necessário verificar as relações entre as classes dos componentes (MARTIN, 2017). Um diagrama de pacotes como ilustrado na Figura 3.3.3.3.1 pode ser útil no cálculo da métrica.

Premissas:

- Ca é o acoplamento aferente, dado pelo número de classes externas que dependem de classes internas ao componente
- Ce é o acoplamento eferente, dado pelo número de classes externas de quem as classes internas dependem

A instabilidade é dada pela fórmula:

$$I = \frac{Ce}{Ca + Ce} \tag{3.1}$$

O valor da instabilidade é um número entre 0 e 1, podendo assim ser medido também através de porcentagem.

#### 3.3.3.3.2 Abstração

O nível de abstração de um componente mede quantas as classes do componente são abstratas.

Premissas

- Na: Número de classes abstratas no componente
- Nc: Número total de classes no componente

A abtração do componente é dada pela fórmula:

$$A = \frac{Na}{Nc} \tag{3.2}$$

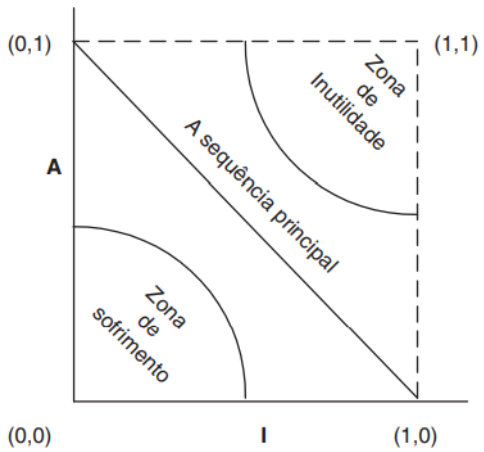
#### 3.3.3.3.3 Sequência Principal

O SDP revela que em uma configuração de componentes ideal apresentaria alguns componentes instáveis, com instabilidade 1, dependendo de um único componente estável, com instabilidade 0. Pelo SAP, nessa configuração ideal os módulos instáveis tem abstração 0, enquanto o módulo estável possui abstração 1.

Pode-se então construir um gráfico A/I plotando-se as métricas de abstração e instabilidade. A partir dos pontos (A=0, I=1), para os módulos instáveis, e (A=1, I=0), para o módulo estável; define-se uma reta denominada **sequência principal**.

Módulos próximos da origem ou próximos do ponto (1,1) não são desejáveis pois são muito abstratas e instáveis, portanto não são utilizadas. Essa região é chamada de **Zona de Inutilidade**. Pontos próximos a origem são extremamente acopladas e concretas. Essa

Figura 12 – Gráfico A/I ilustrando a sequência principal e as zonas de exclusão




Fonte: (MARTIN; MARTIN, 2011)

região costuma sofrer dos maus cheiros de projeto e é chamada de **Zona de sofrimento**. Tanto a zona de sofrimento quanto de inutilidade devem ser evitadas, podendo ser chamadas de **Zonas de Exclusão**. A Figura 12 ilustra o gráfico A/I, a sequência principal e as zonas de exclusão.

Essas definições permitem o cálculo de duas métricas: a distância da sequência principal e a distância normalizada (da sequência principal). Suas fórmulas são, respectivamente:

$$D = \frac{|A + I - 1|}{\sqrt{2}} \tag{3.3}$$

$$D' = |A + I - 1| \tag{3.4}$$

Os intervalos que esse valores podem assumir são, respectivamente:

$$D \in [0, \frac{1}{\sqrt{2}} \approx 0,707] \tag{3.5}$$

$$D' \in [0, 1] \tag{3.6}$$

A métrica de distância normalizada para a sequência principal pode ser bastante útil para a gestão de qualidade de um código. Uma forma dela contribuir na gestão é adotando-se um limiar de controle, e quando o pacote excede esse limiar, deve-se investir tempo para identificar o motivo dessa distância estar além desse limiar (MARTIN; MARTIN, 2011).

### 3.3.4 Arquitetura

Fowler (FOWLER, 2019) define arquitetura de software como a estrutura do *software* que reflete decisões importantes sobre as funcionalidades. Defende também que é importante para que novas funcionalidades demandem menos esforço de desenvolvimento.

Para Jacobson (JACOBSON, 1992), a arquitetura de *software* são estruturas que tem como prioridade dar suporte aos casos de uso de um sistema. Martin (MARTIN, 2017) concorda, afirmando que arquiteturas devem "gritar"casos de uso.

Martin (MARTIN, 2017) diz que boas decisões de arquitetura adiam ao máximo tomadas de outras decisões, como uso de *frameworks*, banco de dados, tecnologias. Ele também afirma que a web é um mero detalhe na construção de um sistema, que tem seu núcleo voltado para os casos de uso e as regras de negócio.

#### 3.3.4.1 Arquitetura Boundary-Control-Entity (BCE)

Jacobson (JACOBSON, 1992) introduziu um modelo de arquitetura que ficou conhecido como arquitetura BCE ou Fronteira-Controlador-Entidade. A Figura 13 ilustra essa arquitetura e seus componentes: ator, fronteira, controlador e entidade.

Nesse modelo de arquitetura, o ator precisa de um meio de comunicar com o sistema. A interface pode ser uma requisição web vinda de um aplicativo móvel, de um site, um comando no terminal etc. Qualquer que seja a tecnologia, haverá um componente, uma fronteira, responsável pela recepção do pedido, encaminhamento ao componente que deverá validar as regras do pedido feito, o controlador, e retornar uma resposta ao ator. Como outros sistemas, como banco de dados, podem ser considerados outros atores, é possível que as fronteiras também façam o intermédio na comunicação entre sistemas.

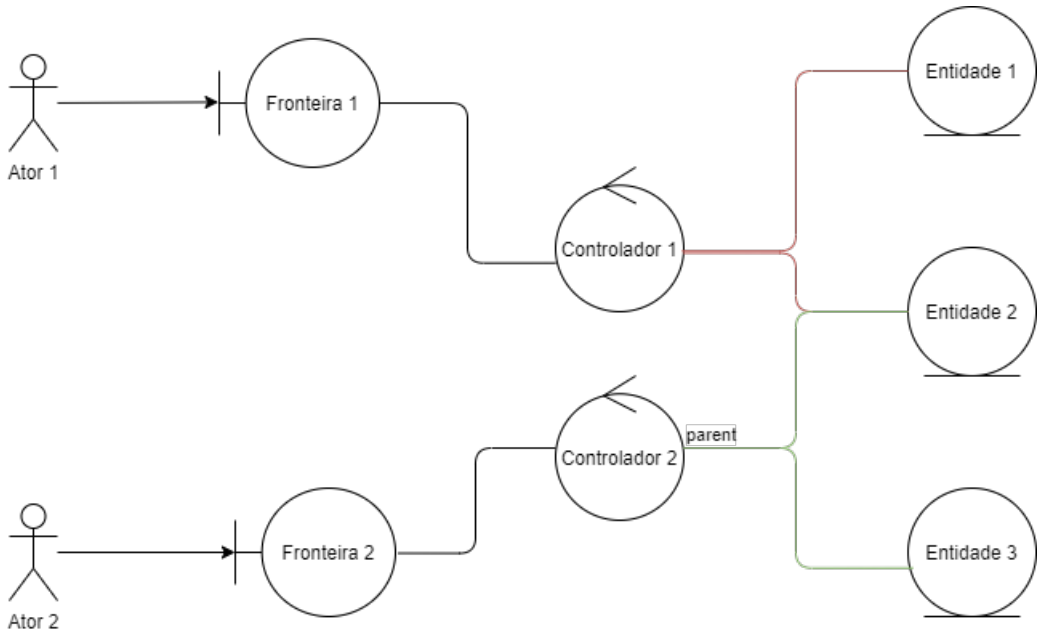
O valor percebido pelo software vem do que (MARTIN, 2017) define como as **regras de negócio** (do inglês, *business rules*), que englobam as abstrações e regras necessárias para a entrega de utilidade ao usuário. Quando atreladas a certos dados críticos ao negócio, passam a integrar o que (JACOBSON, 1992) revela como entidade.

Os casos de uso numa arquitetura BCE podem estar tanto nos controladores quanto nas entidades. Quando estão num controlador, definem as restrições para o uso das entidades.

#### 3.3.4.2 Arquitetura Limpa

Martin (MARTIN, 2017) se inspirou na arquitetura BCE (JACOBSON, 1992), e em outras como a Arquitetura Hexagonal (COCKBURN, 2005) e a DCI (REENSKAUG; COPLIEN, 2009), criando a Arquitetura Limpa unindo os princípios fundamentais dessas arquiteturas. A Figura 14 exibe o modelo da arquitetura limpa. Os objetivos dessa

Figura 13 – Exemplo de software com Arquitetura BCE




Fonte: (PEARCE, 2015)

arquitetura são ser independente de banco de dados, *frameworks*, interfaces, etc além de ser testável.

Nessa arquitetura em camadas, existe uma regra que dita que as dependências devem fluir das camadas exteriores para camadas interiores, nunca ao contrário. No canto inferior direito da Figura 14 existe uma solução que pode ser usada quando houver necessidade de dependência "proibida"pela regra.

Na Figura 14, há uma diferença de notação em relação à da arquitetura BCE: o controlador da arquitura BCE equivale ao Interador de Caso de Uso da Arquitetura Limpa, enquanto o controlador dessa arquitetura se assemelha mais ao da Arquitetura MVC.

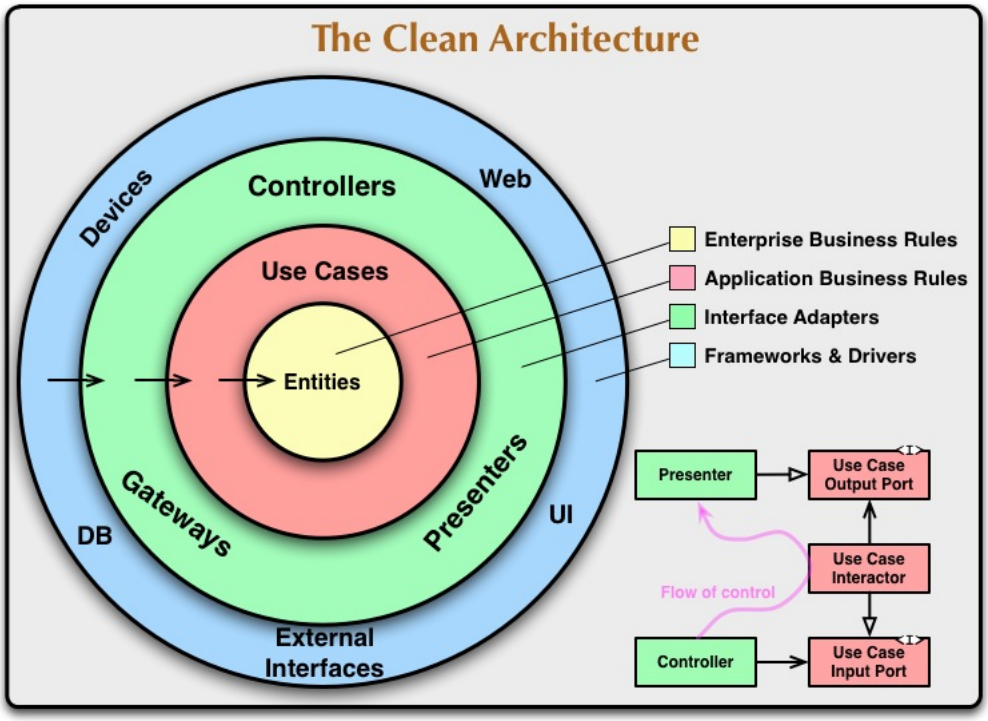
### 3.3.5 *TDD - Test-Driven Design*

Uma dos desafios de desenvolvedores ao desenvolver produtos é dar garantia de que o produto faz corretamente o que ele se propõe a fazer. Existem diversas técnicas de provas matemáticas para provar a corretude e a eficiência de algoritmos (CORMEN; LEISERSON; RIVEST, 2002).

Embora seja correto utilizar de métodos matemáticos para provar a corretude de programas, parte da prova implica em provar que o código efetivamente escrito representa com exatidão o modelo provado pelo método (CORMEN; LEISERSON; RIVEST, 2002). Nessa transição, pequenos detalhes podem facilmente passar despercebidos.

Pensando em ganhar produtividade, diversos autores buscaram outras formas mais

Figura 14 – Modelo de Arquitetura Limpa




Fonte: (MARTIN, 2017)

práticas para garantir que o software funcione de maneira adequada. O desenvolvimento guiado a testes (do inglês, *Test-Driven Design*, ou simplesmente *TDD*) é um método análogo ao método científico para a produção de provas (ELLIOTT, 2020).

O Desenvolvimento Guiado a Testes se dá pelas seguintes fases (BECK, 2002):

1. Vermelha: marcada pela construção de testes, e do código a ser testado. Erros de compilação são comuns nessa fase.
2. Verde: o código deve ser aprimorado até passar nos testes criados
3. Amarela: marcada pela refatoração do código.

Essas fases são feitas em ciclos até que o código tenha a funcionalidade proposta.

Essa abordagem também possui algumas desvantagens. Abaixo está uma lista de possíveis desvantagens.

1. A prova produzida não é tão robusta quanto uma prova matemática. Porém essa mesma característica torna o código mais flexível do que um código provado matematicamente. (ELLIOTT, 2020)

2. *TDD* pode gerar arquiteturas ruins se não utilizar princípios de arquitetura. (MARTIN, 2017)

3. A eficácia de melhoria de código, apesar de ser empiricamente experimentada (MARTIN, 2016a), pode não ser afetada pela ordem da geração do teste e do código, desde que ambos existam ao final do processo (FUCCI et al., 2016).

Em contrapartida, a disciplina de testes também apresenta vantagens importantes. Uma das vantagens do TDD mais defendidas por (MARTIN; MARTIN, 2011) é a naturalidade que a filosofia tem de fazer software que tenha baixos níveis de acoplamento, o que os autores em questão defendem como sendo mais prejudicial a sistemas que passam por muitas evoluções.

Apesar de suas desvantagens, a filosofia *TDD*, que prega que os testes devem vir antes do desenvolvimento do código, tem sido bastante maturada na comunidade. Também serviu de base para outras inovações (ALLIANCE, 2020), como o *BDD*, Desenvolvimento Guiado a Comportamento (do inglês, *Behavior-Driven Design*) introduzido por (NORTH, 2006).

### 3.3.6 API

O termo API é um acrônimo para *Application Programming Interface* (do inglês, Interface de Programação de Aplicações). (DAVIDSE, 2020) define essa expressão como "um conjunto de definições e protocolos que permitem que duas aplicações conversem entre si". Essa conversa é feita através pela troca de informações, que é compreendida no contexto da interface.

APIs podem ser utilizadas por diferentes aplicações, suportando públicos diferentes. Consequentemente, há uma classificação que identifica a API pelo seu público (HAT, 2021):

- API Pública

  Usada por desenvolvedores terceiros

- API de Parceiros

  Usada por aplicações de parceiros de negócio específicos

- API Privada

  Usada por aplicações internas

A construção de APIs é balizada por regras e protocolos que são aplicados ao fazer chamadas de API. Esta, por sua vez, ocorre quando informação flui através de um *endpoint* (DAVIDSE, 2020).

Existem três principais tipos de API utilizadas no contexto de *software web* (DAVIDSE, 2020):

- RPC

  Introduzido por (NELSON, 1981), RPC ou Chamada de Procedimento Remoto (do inglês, *Remote Procedure Call*) é um protocolo de requisição e resposta. Foi o primeiro tipo de API amplamente adotado, mas que hoje é considerado antiquado devido ao custo de criação e manutenção de API mais elevado que em outras abordagens (DAVIDSE, 2020).

- SOAP

  Introduzido por (WINER, 1998), SOAP ou Protocolo de Acesso a Objetos Simples (do inglês, *Simple Object Access Protocol*) é um protocolo oficial mantido pela World Wide Web Consortium W3C (GUDGIN et al., 2007) criado para ser uma especialização do RPC utilizando mensagens codificadas em XML. Seu uso foi reduzido após a introdução do REST (DAVIDSE, 2020).

- REST

  Introduzido por (FIELDING; TAYLOR, 2000), REST ou Transferência Representacional de Estado (do inglês, *REpresentational State Rransfer*) define um protocolo onde as mensagens podem ser codificadas em JSON, XML etc. Tecnologia se tornou líder de mercado (MOTROC, 2017) por ser considerada mais simples de usar, quando comparado com SOAP (RODRIGUEZ, 2008).

#### 3.3.6.1 REST

(RODRIGUEZ, 2008) defende que *APIs REST* devem seguir ao menos os seguintes quatro critérios:

1. Usar verbos HTTP explicitamente
2. Não apresentar estado
3. Expor *URIs* semelhantes a estrutura de diretórios
4. Objetos de transferência em XML, JSON ou ambos

Destaca-se do primeiro critério a existência de um mapeamento entre as chamadas operações CRUD (*Create Read Update Delete*, ou criação, leitura, atualização e remoção, em português) e verbos HTTP. Esse mapeamento se dá da seguinte forma (RODRIGUEZ, 2008):

- Criação - use *POST*
- Leitura - use *GET*
- Atualização - use *PUT*
- Remoção - use *DELETE*

Apesar dessa recomendação ser válida, desenvolvedores de APIs REST precisam estar conscientes que a utilização do método *GET* implicar em expor a aplicação a alguns riscos. Como esse método não possui um corpo de requisição onde se pode passar argumentos, a passagem de argumentos é feita pela *query string*.

Quando uma requisição envolve algum dado sensível se utiliza do método *GET*, o dado está acessível para qualquer pessoa que tenha acesso à *query string*, o que configura uma vulnerabilidade do sistema (COMMUNITY, 2020). Uma possível solução para o problema é utilizar um método *POST* para leitura de um recurso, uma vez que este método é capaz de encapsular parâmetros através do corpo de sua requisição.

(FOWLER, 2010) defende que o processo de criação de uma API possui os seguintes níveis de maturidade:

0. A lama de XML
1. Recursos
2. Verbos HTTP
3. Controle Hipermídia
4. Glória do REST

# 4 Desenvolvimento

## 4.1 Planejamento inicial

### 4.1.1 O time do cliente

Não é possível realizar desenvolvimento efetivo sem que o desenvolvedor saiba o que irá desenvolver, então é sensato que se inicie o projeto realizando uma definição preliminar do produto proposto. Como metodologias ágeis sempre optam pela maior participação possível das partes interessadas no sistema (COHN, 2004), a primeira coisa que foi realizada foi abordar informalmente, em conversa, alguns usuários em potencial.

Na seção 2.1, foi proposto que o time de cliente seria composto por 4 pessoas, sendo que 2 dessas pessoas seriam médicos. Infelizmente, os 2 médicos que participariam dessa equipe de trabalho puderam apenas contribuir em conversas iniciais pois eles contraíram COVID-19 e se retiraram devido às suas complicações. Felizmente, a recuperação foi bem-sucedida, porém não ocorreu a tempo de agregar substancialmente a esse trabalho.

Para viabilizar o trabalho, duas pessoas que não trabalham no ramo da saúde foram agregadas ao time de cliente. (COHN, 2004) relata que cada perfil de usuário enviesa um projeto ao expor suas expectativas, e é de se esperar que a remoção de um dos perfis de usuário levaria o projeto ao viés de seus demais usuários.

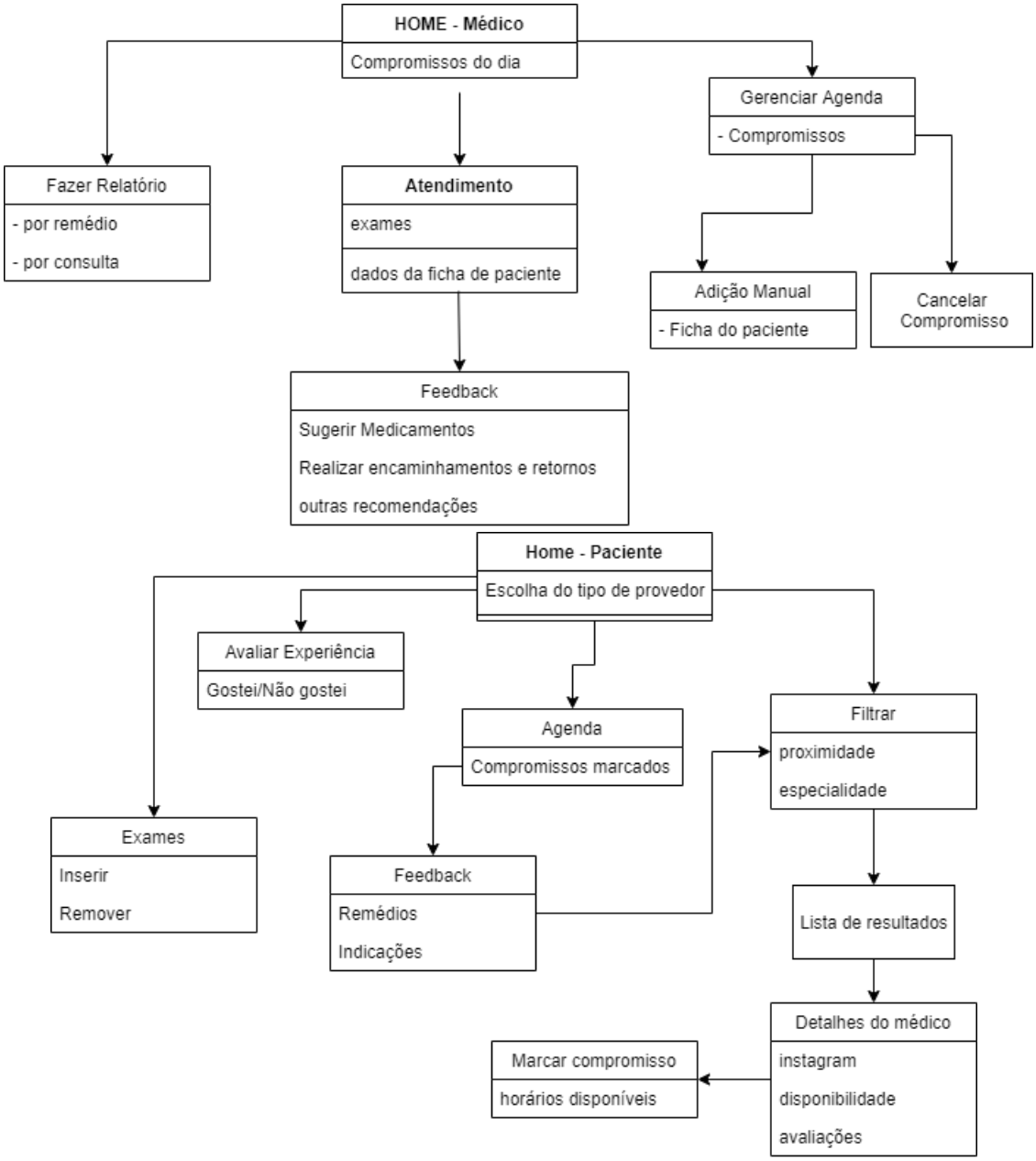
### 4.1.2 Escrita de histórias

Uma vez montado o time do cliente, foi conduzida uma sessão de escrita de histórias. Essa sessão foi iniciada com uma conversa, alinhando com todos o objetivo inicial do software, que é auxiliar médicos e pacientes no agendamento de consultas, possibilitando que pacientes em potencial possam também descobrir e avaliar esses profissionais.

Ainda nas discussões iniciais sobre as funcionalidades básicas para o sistema, levantou-se a possibilidade de o sistema dar suporte ao atendimento médico de forma remota, a chamada telemedicina. Porém, foi notado que a legislação brasileira traz algumas restrições para o funcionamento dessa funcionalidade, como proposto pela (MEDICINA, 2018) e (MEDICINA, 2002). Mesmo assim, a legislação ainda se mostra volátil pela recente revogação da (MEDICINA, 2018) pela (MEDICINA, 2019). Dada essa instabilidade e burocracia, optou-se por remover essa funcionalidade do escopo inicial do projeto.

Em seguida, a reunião passou a permitir que cada participante pensasse em funcionalidades que o sistema proposto poderia ter. Esse *brainstorm* inicial foi-se maturando, até que no terceiro momento da sessão, foi desenvolvido um mapa de *site* para o sistema,

Figura 15 – Mapa do *site* do produto proposto




O desenho revela dois tipos de experiências para o produto: para o médico, na figura de cima, e para o paciente na figura de baixo.

exibido na figura 15. Nele, existem dois pontos de partida, uma *Home* para o médico, e uma para o paciente. Cada tela é representada por um retângulo, sendo a parte inferior responsável por representar informações contidas nessa tela. As setas representam as ações que cada usuário poderá tomar, mudando de tela.

Outro ponto levantado na criação do protótipo foi que o sistema poderia incluir atividades do ramo da saúde que não são médicas propriamente ditas, como terapias, tratamentos, etc. A partir desse momento, definiu-se que um dos atores principais não seria necessariamente um médico, mas um provedor de saúde, que poderia ser um médico,

ou outro profissional também da área da saúde.

O mapa do *site* foi utilizado para a escrita de histórias de usuário. Essas histórias tiveram sua complexidade de desenvolvimento estimada e a partir dessa estimativa, o momento final da sessão foi responsável pela ordenação da importância a partir da prioridade para o time de clientes. As histórias, já em ordem, são:

1. Um paciente pode encontrar um provedor de saúde em uma lista e analisá-lo
2. Um paciente pode marcar, desmarcar e verificar compromissos com um provedor de saúde
3. Um provedor de saúde pode ver seus compromissos e cancelá-los
4. Um provedor de saúde pode ver dados disponíveis do paciente durante o atendimento
5. Um paciente pode inserir e remover exames no sistema
6. Um provedor de saúde pode adicionar compromissos e clientes manualmente
7. Um paciente pode avaliar um provedor de saúde
8. Um provedor de saúde pode solicitar relatórios de seus atendimentos
9. Um provedor de saúde pode customizar seu perfil
10. Um paciente pode receber *feedback* de um atendimento com um provedor de saúde
11. Um provedor de saúde pode fornecer *feedback* do atendimento

## 4.2 Gestão do Projeto

A frase abaixo é atribuída a Dr[a] Pamela Zave: (JASPAN; SADOWSKI, 2019)

> O propósito da engenharia de *software* é gerenciar a complexidade, não criá-la.

Uma frase popular no meio administrativo mas falsamente atribuída a Peter Drucker (ZAK, 2013) dizia, por sua vez, que:

> O que não pode ser medido não pode ser gerenciado.

As frases anteriores sugerem que uma tarefa importante da engenharia de software é a medida de complexidade. Como a partir dela podem ser tomadas de decisões relevantes para o gerenciamento do projeto, esse aspecto pode ser considerado um KPI.

A unidade utilizada para estimar a complexidade é uma medida nebulosa de tempo (ou NUT, do ingês *Nebulous Unit of Time* (COHN, 2004)) que frequentemente é denominada por pontos, onde cada ponto representa uma hora de trabalho. A distribuição escolhida para classificar cada história é a dos múltiplos de 4, por facilitar a conversão da estimativa em dias, uma vez que 8 pontos representam um dia de trabalho para uma pessoa.

O critério da avaliação da complexidade deveria avaliar todo o processo para a entrega do valor até o cliente (COHN, 2004). Nisso, se imaginou que o sistema funcionaria em um aplicativo *mobile*, com as regras de negócio sendo executadas em algum servidor *web*.

Cada história teve sua complexidade avaliada. Seus valores se encontram na Tabela 1. Verificando alguma comunalidade em histórias adjacentes, criou-se quatro pacotes que oferecem funcionalidades entregáveis. Cada pacote recebeu as histórias de usuário como também indicado pela Tabela 1. Os quatro pacotes são:

1. Agendamento
2. Atendimento
3. Suporte ao provedor
4. *Feedback*

A partir desses valores de complexidade, e organização da estrutura de pacotes, desenvolveu-se um cronograma utilizando o diagrama de Gantt, para avaliar se seria possível a realização do projeto. Esse diagrama pode ser visto na Figura 16.

O uso dessa ferramenta de análise revelou que a existência da restrição de apenas um recurso de desenvolvimento impossibilitaria o paralelismo de execução de tarefas. Esse comportamento sequencial das atividades faria o desenvolvimento durar até o dia 20 de Abril de 2021, sem ter entregado sequer o primeiro pacote entregável antes de Março de 2021.

Como o *deadline* do projeto é deveras anterior a essa data, a gestão do projeto deveria tomar uma decisão para lidar com essa situação. Optou-se em restringir o escopo do projeto apenas à aplicação servidora, tendo um protótipo de interface em alta fidelidade para meros fins de ilustração e sem a aplicação de um banco de dados real.

Diante da tomada desta decisão, houve duas possibilidades: recomeçar a análise de complexidade, ou assumir que metade do trabalho inicial seria utilizado para a construção de uma aplicação *front-end*, e isso poderia ressignificar cada ponto. A segunda opção foi a escolhida, com uma razão onde cada 2 pontos passariam a significar 1 hora de trabalho de

| História de usuário | Complexidade | Entregável | Apelido |
|---|---|---|---|
| Um paciente pode encontrar um provedor de saúde em uma lista e analisá-lo | 80 | Agendamento | US01 |
| Um paciente pode marcar, desmarcar e verificar compromissos com um provedor de saúde | 80 | Agendamento | US02 |
| Um provedor de saúde pode ver seus compromissos e cancelá-los | 80 | Agendamento | US03 |
| Um provedor de saúde pode ver dados disponíveis do paciente durante o atendimento | 20 | Atendimento | US04 |
| Um paciente pode inserir e remover exames no sistema | 40 | Atendimento | US05 |
| Um provedor de saúde pode adicionar compromissos e clientes manualmente | 28 | Suporte ao provedor | US06 |
| Um paciente pode avaliar um provedor de saúde | 20 | Suporte ao provedor | US07 |
| Um provedor de saúde pode solicitar relatórios de seus atendimentos | 40 | Suporte ao provedor | US08 |
| Um provedor de saúde pode customizar seu perfil | 28 | Suporte ao provedor | US09 |
| Um paciente pode receber *feedback* de um atendimento com um provedor de saúde | 40 | *Feedback* | US10 |
| Um provedor de saúde pode fornecer *feedback* do atendimento | 56 | *Feedback* | US11 |

Tabela 1 – Estimativas de complexidade, a quais pacote entregáveis elas pertencem e os apelidos das histórias de usuário.

um único recurso. A vantagem dessa escolha foi de não precisar reavaliar os valores de complexidade, mantendo assim os valores descritos na Tabela 1.

A partir das decisões tomadas, foi possível refazer o cronograma de entrega utilizando-se novamente o diagrama de Gantt, como pode ser visto na Figura 17. A partir dessa análise, vê-se a viabilidade de se entregar pelo menos o primeiro pacote entregável a tempo para o *deadline*.

### 4.2.1 Gerenciamento de Iteração

Metodologias ágeis trabalham com ciclos de entrega envolvendo o cliente no desenvolvimento, chamados de iterações (ou *sprints*, em inglês). Como tais metodologias focam no trabalho em equipe, faz sentido em se fixar os períodos de duração, normalmente em 2 ou 3 semanas, para que a performance nesse período possa ser mensurada (COHN, 2004).

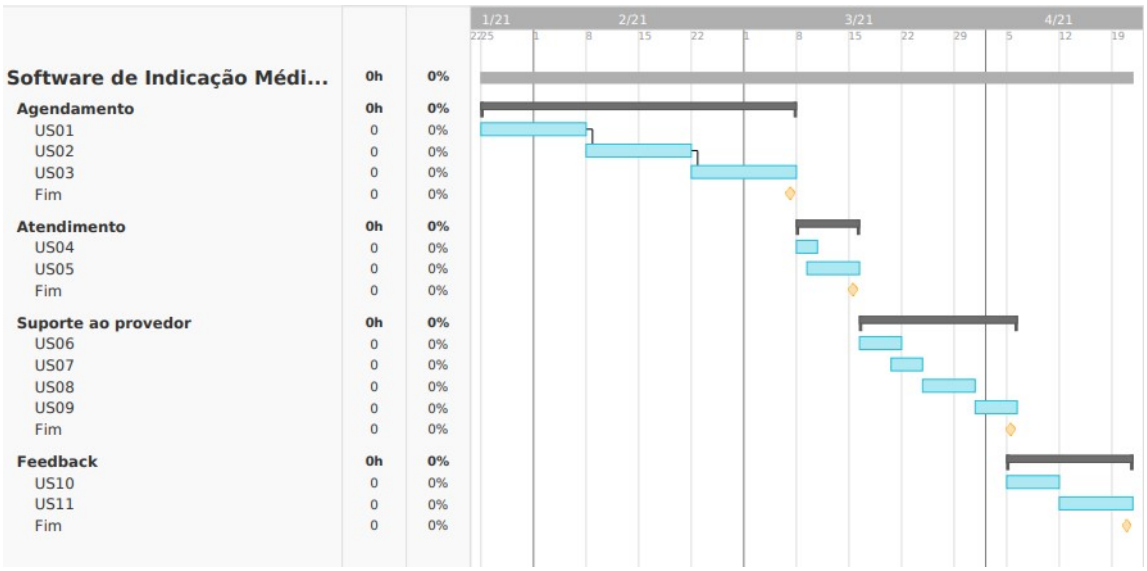



Figura 16 – Diagrama de Gantt com as estimativas iniciais

Estimativas se aplicam a todos os pacotes e histórias de usuário.

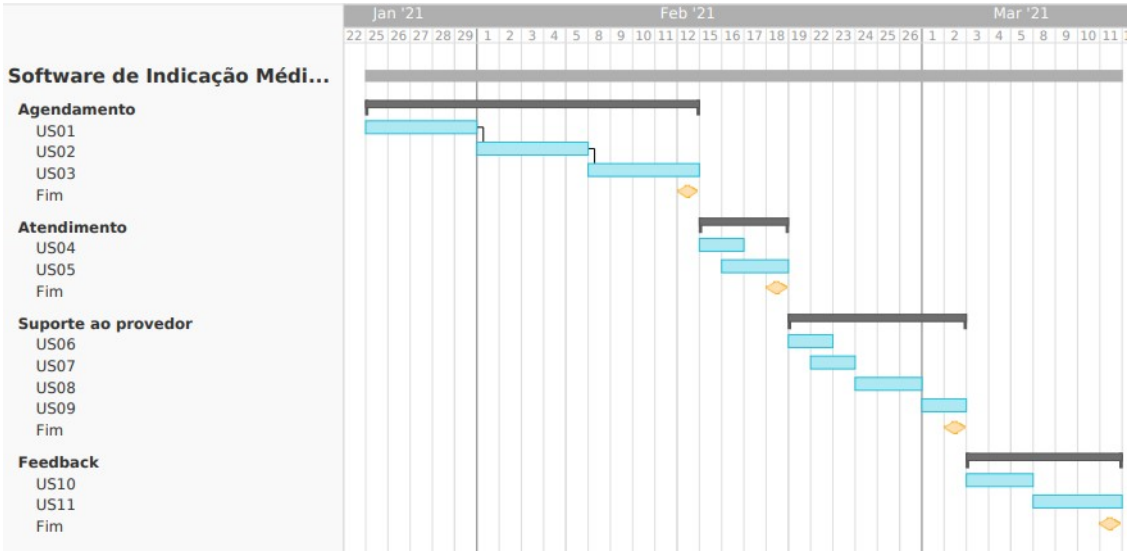



Figura 17 – Diagrama de Gantt com as estimativas finais

Estimativas se aplicam a todos os pacotes e histórias de usuário, feitas com o novo significado de pontos.

Quando se está trabalhando como o único recurso desenvolvedor do projeto, os desafios do trabalho em equipe são naturalmente mitigados. Todavia, o planejamento e os indicadores de performance ainda são necessários.

Optou-se por uma solução incomum para a duração de cada iteração, em vez dela ser definida em períodos fixos, passa a ser variável. Essa escolha foi feita de modo a equilibrar a carga de planejamento e desenvolvimento. Como até a *deadline* poderia haver cerca de 2 entregas, escolheu-se que cada período de iteração seria o período de desenvolvimento de cada entrega.

Como os métodos ágeis possuem diversas reuniões para garantir qualidade na produção de código (COHN, 2004), essa solução adota nesse projeto de um único indivíduo se aplicada em um time de desenvolvimento poderia levar ao malefício de possivelmente reduzir a qualidade do desenvolvimento. Isso por que mesmo em metodologias ágeis, pode-se adotar métodos de garantia de qualidade como o PDCA, além da filosofia *Kaizen*.

# 5 Resultados

## 5.1 Primeira entrega: Agendamento

Na Tabela 1 é mostrada a composição das *user stories* para todos os pacotes do projeto. Para o primeira entrega, as seguintes histórias de usuário estão inclusas:

- US01 - Um paciente pode encontrar um provedor de saúde em uma lista e analisá-lo;
- US02 - Um paciente pode marcar, desmarcar e verificar compromissos com um provedor de saúde;
- US03 - Um provedor de saúde pode ver seus compromissos e cancelá-los.

Ao longo da iteração, notou-se que duas dessas histórias necessitariam ser subdivididas. Portanto, essas histórias que foram subdivididas podem ser considerados épicos.

O primeiro épico, nomeado de EP01, se trata da própria US02. Ele é composto pelas novas histórias de usuário:

- US12 - Um paciente pode marcar um compromisso com um provedor de saúde;
- US13 - Um paciente pode verificar compromissos com um provedor de saúde;
- US14 - Um paciente pode desmarcar um compromisso com um provedor de saúde.

O segundo épico, nomeado de EP02, se trata da própria US03. Ele é composto pelas seguintes histórias de usuário:

- US15 - Um provedor pode ver seus compromissos;
- US16 - Um provedor de saúde pode cancelar seus compromissos.

### 5.1.1 Casos de Uso

Para (MARTIN, 2017), os casos de uso são a parte mais importante de um sistema. Para ele, uma boa arquitetura de um sistema deve "gritar"seus casos de uso.

Ao longo da iteração, cada iteração foi inicialmente desenvolvida como um caso uso próprio. A Figura 18 mostra o caso de uso da US01, assim como a Figura 19 representa o caso de uso da US12.

Todavia, após o desenvolvimento das US13 e US14, percebeu-se que as histórias US15 e US16 eram, respectivamente, semelhantes. (HUNT; THOMAS, 1999) defendem que no desenvolvimento de sistemas, o programador não deve se repetir, o que os autores chamam de princípio *DRY - Don't Repeat Yoursel* (do inglês, Não Se Repita), altamente disseminado pela Programação Extrema (MARTIN; MARTIN, 2011).

Para ser possível a não repetição, elevou-se o grau de abstração. As US13 e US15 não teriam um caso de uso para cada ator fazer o mesmo; mas um terceiro ator, o Usuário, que poderia ser um Paciente ou um Provedor, passaria a interagir com o caso de uso. O resultado dessa abstração são os casos de uso representados na Figura 20, referente às US13 e US15, e na Figura 21, com estratégia análoga e referente às US14 e US16.

Figura 18 – Caso de Uso para a US01

Ator Primário Paciente

Escopo Agendamento

Cenário de sucesso principal

1. Usuário insere os critérios de busca, podendo ser distância máxima ou especialidade
2. Sistema busca os provedores e retira os que não atendem aos critérios
3. O sistema retorna ao usuário uma lista detalhada dos provedores

Extensões

2a. Caso o sistema não encontre provedores, o resultado é uma lista vazia

2b. Caso não haja critérios de busca, o resultado é uma lista com todos os provedores

Figura 19 – Caso de Uso para a US12

Ator Primário Paciente

Escopo Agendamento

Cenário de sucesso principal

1. Paciente escolhe o provedor e o horário
2. Sistema verifica a disponibilidade do médico e do paciente
3. Sistema adiciona compromisso à agenda do provedor
4. Sistema adiciona compromisso à agenda do paciente

Extensões

2a. Caso o médico não exista, ou não haja disponibilidade, é mostrado um erro ao usuário

Figura 20 – Caso de Uso para a US13 e US15

Ator Primário Usuário

Escopo Agendamento

Cenário de sucesso principal

1. Usuário informa sua identificação
2. Sistema verifica os compromissos marcados
3. Sistema verifica os nomes dos demais participantes
4. Sistema retorna a lista dos compromissos com os nomes e horários

Figura 21 – Caso de Uso para a US14 e US16

Ator Primário Usuário

Escopo Agendamento

Cenário de sucesso principal

1. Usuário informa sua identificação e o horário de compromisso a ser cancelado
2. Sistema busca os dados do compromisso e do usuário par ao compromisso
3. Sistema remove o compromisso do par
4. Sistema remove o compromisso do usuário

Extensões

2a. Caso dados não sejam encontrados, é apresentado um erro ao usuário

### 5.1.2 Testes de Aceitação

Cada história de usuário demanda seu próprio teste de aceitação, ainda que seus casos de uso sejam mesclados. São eles:

- US01 - Um paciente pode encontrar um provedor de saúde em uma lista e analisá-lo:
  - Testar com raio menor que a distância para qualquer provedor e não encontrar;
  - Testar com raio intermediário e verificar exclusão dos provedores fora do raio;
  - Buscar cirurgiões e apenas listar cirurgiões;
  - Buscar cirurgiões dentro de um raio, e apenas listar os cirurgiões dentro desse raio de distância;
  - Não inserir critérios deverá listar todos os médicos;
  - Busca deve revelar nome, mídia social, especialização do provedor.
- US12 - Um paciente pode marcar um compromisso com um provedor de saúde:

  - Marcar apenas uma consulta por hora;
  - Horário disponível deverá ser das 9 às 18, com horário de almoço às 12:00;
  - Marcar fora do horário disponível deve falhar;
  - Marcar em horário já preenchido do provedor deve falhar;
  - Marcar em horário já preenchido do cliente deve falhar.

- US13 - Um paciente pode verificar compromissos com um provedor de saúde:
  - Paciente com 3 compromissos deve exibir todos os compromissos e os nomes dos provedores devem estar corretos;
  - Paciente não existente deve falhar;
  - Provedor do compromisso não existente deve falhar.

- US14 - Um paciente pode desmarcar um compromisso com um provedor de saúde:
  - Tentar com um paciente que tem um compromisso com um provedor no horário do compromisso e passar;
  - Tentar com um paciente que tem um compromisso com um provedor que não tem compromisso de volta e falhar;
  - Tentar com um paciente que não tem compromissos e falhar.

- US15 - Um provedor pode ver seus compromissos:
  - Provedor com 3 compromissos deve exibir todos os compromissos e os nomes dos pacientes devem estar corretos;
  - Provedor não existente deve falhar;
  - Paciente do compromisso não existente deve falhar.

- US16 - Um provedor de saúde pode cancelar seus compromissos:
  - Tentar com um provedor que tem um compromisso com um paciente no horário do compromisso e passar;
  - Tentar com um provedor que tem um compromisso com um paciente que não tem compromisso de volta e falhar;
  - Tentar com um provedor que não tem compromissos e falhar.

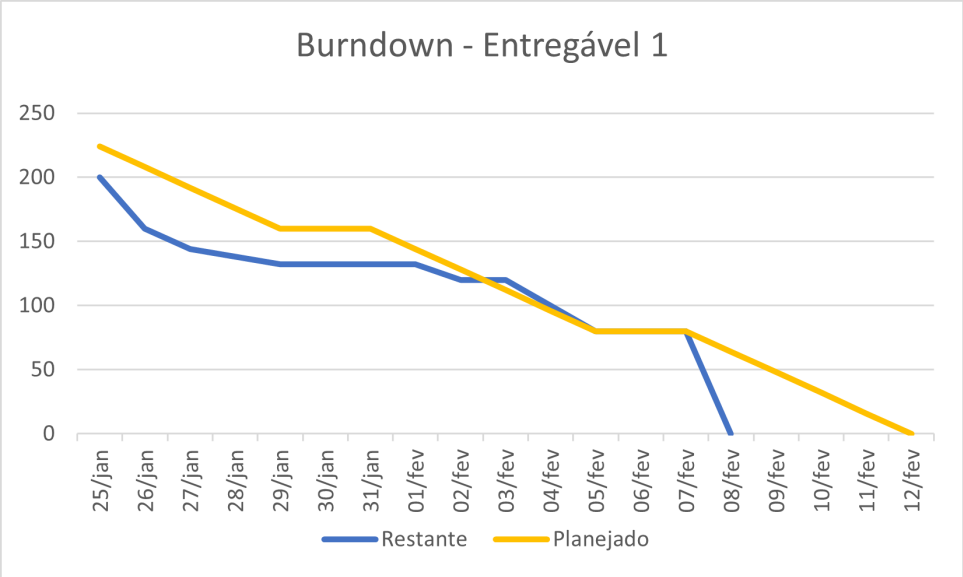
### 5.1.3 Performance de Desenvolvimento

Na Figura 22, pode-se ver o gráfico de *burndown* do primeiro entregável. Nele, observa-se que até o dia 2 de Fevereiro de 2021, o projeto esteve adiantado. Entre os dias

2 de Fevereiro e 5 de Fevereiro, esteve em atraso, tendo esse atraso zerado ao final desse período.

Entre os dias 7 e 8 de Fevereiro, houve um adiantamento significativo, culminando em um fim adiantado da iteração. Esse adiantamento é explicado pela junção dos casos de uso das US13 e US15, além da junção dos casos de uso da US14 e US16.

Figura 22 – Gráfico de *Burndown* do primeiro entregável




Em azul, os pontos restantes avaliados após o dia utilizado no desenvolvimento. Em amarelo, os pontos restantes ao final do dia esperados pelo planejamento. Nesse tipo de gráfico, quando a curva restante está acima da curva amarela, trata-se de um atraso; quando está abaixo, um adiantamento.

## 5.2 Segunda entrega: Atendimento

Da Tabela 1, vemos que as seguintes histórias de usuário estão inclusas:

- US04 - Um provedor de saúde pode ver dados disponíveis do paciente durante o atendimento
- US05 - Um paciente pode inserir e remover exames no sistema

Assim como na primeira iteração, houve a necessidade de transformar histórias de usuário em épicos. Desta vez, apenas a US05 foi desmembrada nas seguintes histórias:

- US17 - Um paciente pode inserir exames no sistema;
- US18 - Um paciente pode remover exames no sistema;

- US19 - O provedor e o paciente podem ver os exames do paciente.

Essa ordenação só foi possível devido à não necessidade da implementação de uma interface gráfica. Não fosse o caso, a US19 teria que possuir prioridade em relação à US18 pois não faria sentido, em termos de usabilidade do usuário, o paciente remover os exames sem sequer conseguir vê-los.

### 5.2.1 Testes de Aceitação

Os testes de aceitação para as histórias integrantes do segundo pacote entregável, responsável pela funcionalidade de Agendamento, são:

- US04 - Um provedor de saúde pode ver dados disponíveis do paciente durante o atendimento:
  - Testar com a hora de um compromisso e retornar o nome, sexo e idade do paciente;
  - Testar com uma hora sem compromissos e falhar;
  - Testar com um provedor sem compromissos e falhar;
  - Testar com um compromisso sem paciente e falhar.
- US17 - Um paciente pode inserir exames no sistema:
  - Testar sem um arquivo válido e falhar;
  - Testar sem um paciente válido e falhar;
  - Testar com um paciente válido e um arquivo válido e passar.
- US18 - Um paciente pode remover exames no sistema:
  - Testar em um horário que possui um compromisso válido e passar;
  - Testar em um horário que não possui um compromisso e falhar;
  - Testar em um horário que possui um compromisso com um provedor inexistente e falhar;
  - Testar com um paciente inválido e falhar.
- US19 - O provedor e o paciente podem ver os exames do paciente:
  - Testar atendimento de um provedor, receber os exames do paciente junto com suas informações;
  - Testar com um paciente com exames, ver seus exames;
  - Testar com um paciente sem exames e receber uma lista vazia.

As restrições de escopo do projeto tiveram algumas consequências na US17. A primeira é como se daria a entrega dos arquivos, que dependeria de uma implementação de entrega, que está fora do escopo. A segunda é o que fazer com os arquivos em questão, uma vez que armazená-los em um banco de dados significaria adotar, pelo menos para os arquivos, um sistema de banco de dados não relacional.

Para (MARTIN, 2017), um bom arquiteto maximiza a quantidade de decisões não tomadas. Como as questões levantadas são intimamente ligadas à entrega de informações da aplicação, decidiu-se que a US17 trabalharia com uma identificação de arquivo em texto, que poderia representar uma identificação de um banco de dados ou o nome do arquivo em si. A US17 adiciona os exames acrescentando essa identificação ao paciente.

Vale ainda destacar que a US19 altera o funcionamento da US04. Essa alteração se dá em razão do primeiro teste adicionar os exames aos dados do paciente acessíveis pelo provedor durante o atendimento.

### 5.2.2 Casos de Uso

Figura 23 – Caso de Uso para a US19

Ator Primário Paciente

Escopo Atendimento

Cenário de sucesso principal

1. Paciente insere identificação
2. Sistema busca o paciente
3. Sistema extrai nome, idade, sexo e os exames do paciente sem alterá-lo
4. Sistema retorna os dados do paciente

Extensões

2a. Caso o sistema não encontre o paciente, é mostrado um erro ao usuário

Embora no início houvesse uma tentativa de fazer casos de uso independentes para cada uma dessas *user stories*, ao longo do desenvolvimento foi visto uma necessidade de uma alteração para obedecer ao princípio *DRY*. Foi o caso da US04, que pelo motivos já relatados reusa a US19. O caso de uso da US04 está escrito em 24, enquanto da US19 está representado na 23.

As considerações sobre o gerenciamento de arquivos de exames, discutidas na última seção tiveram efeitos sob as US17 e US18. Os casos de uso dessas histórias estão representados, respectivamente, pela Figura 25 e pela Figura 26.

Figura 24 – Caso de Uso para a US04

Ator Primário Provedor

Escopo Atendimento

Cenário de sucesso principal

1. Provedor insere identificação e o horário
2. Sistema busca os compromissos do provedor
3. Executa o caso de uso da US19 inserindo a identificação do paciente do compromisso
4. Sistema retorna os dados do paciente

Extensões

2a. Caso o sistema não encontre o provedor ou o compromisso é apresentado um erro

3a. Caso o compromisso não possua paciente, é apresentado um erro

Figura 25 – Caso de Uso para a US17

Ator Primário Paciente

Escopo Atendimento

Cenário de sucesso principal

1. Paciente fornece sua identificação, o identificador de um arquivo (por exemplo, um nome de arquivo ou um link) e um título
2. Sistema busca paciente
3. Sistema cria um registro do exame do paciente com o identificador de arquivo e título
4. Sistema adiciona registro ao registro do paciente

Extensões

2a. Caso o sistema não encontre o paciente é apresentado um erro

3b. Caso o identificador do arquivo esteja vazio é apresentado um erro

Figura 26 – Caso de Uso para a US18

Ator Primário Paciente

Escopo Atendimento

Cenário de sucesso principal

1. Paciente fornece sua identificação e o identificador de arquivo do exame
2. Sistema busca paciente e exame
3. Sistema remove exame do registro do paciente

Extensões

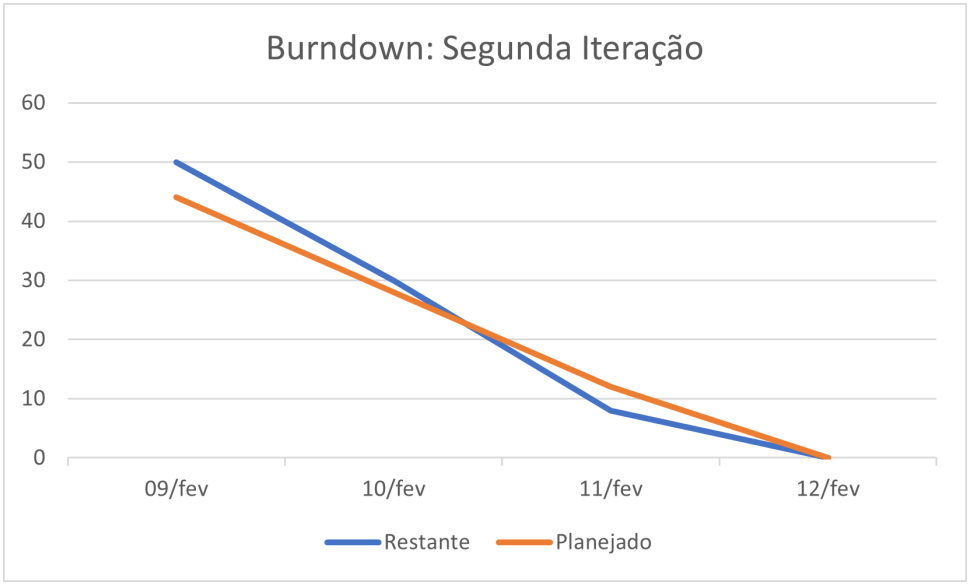
2a. Caso o sistema não encontre o paciente é apresentado um erro

2b. Caso o identificador do arquivo esteja vazio é apresentado um erro

### 5.2.3 Performance de Desenvolvimento

A segunda iteração foi mais curta que a primeira, além de feita a tempo de adentrar neste trabalho devido ao adiantamento da primeira iteração. Para medir a performance ao longo da iteração em si, pode ser usado o gráfico de *burndown* ilustrado pela Figura 27. Através dele, percebe-se que a iteração durou 4 dias, sendo os 2 primeiros marcados por leve atraso, o terceiro por um adiantamento e o último chegando no prazo planejado.

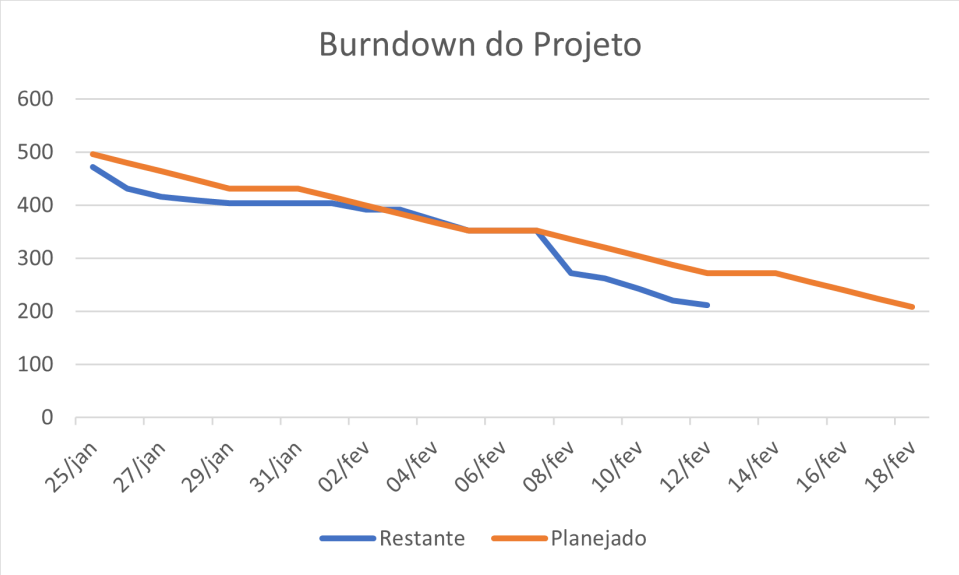
Figura 27 – Gráfico de *Burndown* do segundo entregável




Em azul, os pontos restantes do pacote avaliados após o dia utilizado no desenvolvimento. Em vermelho, os pontos restantes ao final do dia esperados pelo planejamento.

Outra análise possível com a mesma ferramenta é a performance de desenvolvimento do projeto. Essa análise é feita na Figura 28 e mostra como foi importante o adiantamento da primeira iteração para a entrega da segunda iteração.

Figura 28 – Gráfico de *Burndown* da execução do projeto até o segundo entregável




Em azul, os pontos restantes do projeto avaliados após o dia utilizado no desenvolvimento. Em vermelho, os pontos restantes ao final do dia esperados pelo planejamento.

## 5.3 Protótipo de Alta Fidelidade

Para ilustrar a interação entre o homem e o software proposto, foi desenvolvido um protótipo de alta fidelidade. Esse tipo de protótipo é uma representação próxima do que o produto poderia ser, e é uma ótima ferramenta para validar os conceitos envolvidos no produto (IBRAGIMOVA, 2016).

O produto se propõe a oferecer experiências específicas para cada perfil de usuário, seja paciente ou provedor de saúde. Ao iniciar a usabilidade, representado na Figura 29, é possível o usuário selecionar qual das experiências deseja possuir.

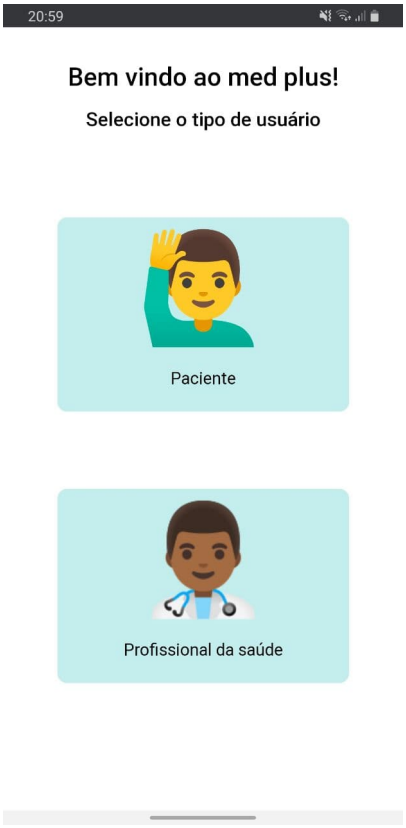
Selecionando para ter a experiência como paciente, há algumas possibilidades a serem utilizadas. A primeira, representada na Figura 30, é a busca e filtragem de provedores de saúde.

A segunda opção está representada na Figura 31. Para aparecer a opção de cancelar, é preciso selecionar um dos compromissos de data futura à da utilização. A figura ainda ilustra a marcação de novos compromissos.

Outra opção para o paciente se dá pela, adição, remoção e visualização de exames no sistema. Essa usabilidade do sistema está representada na Figura 32. Para tornar a visualização genérica o suficiente, ao visualizar um exame basta baixá-lo e utilizar o visualizador padrão para o formato do exame.

Uma vez já tendo participado de uma consulta, o paciente não poderá mais cancelar o compromisso. Todavia, ele ganha a possibilidade de ver recomendações dadas na consulta,

Figura 29 – Tela Inicial do Protótipo de Alta Fidelidade




A tela divide as experiências do paciente e do provedor de saúde

além de avaliar a sua experiência. Essa usabilidade está representada pela Figura 33.

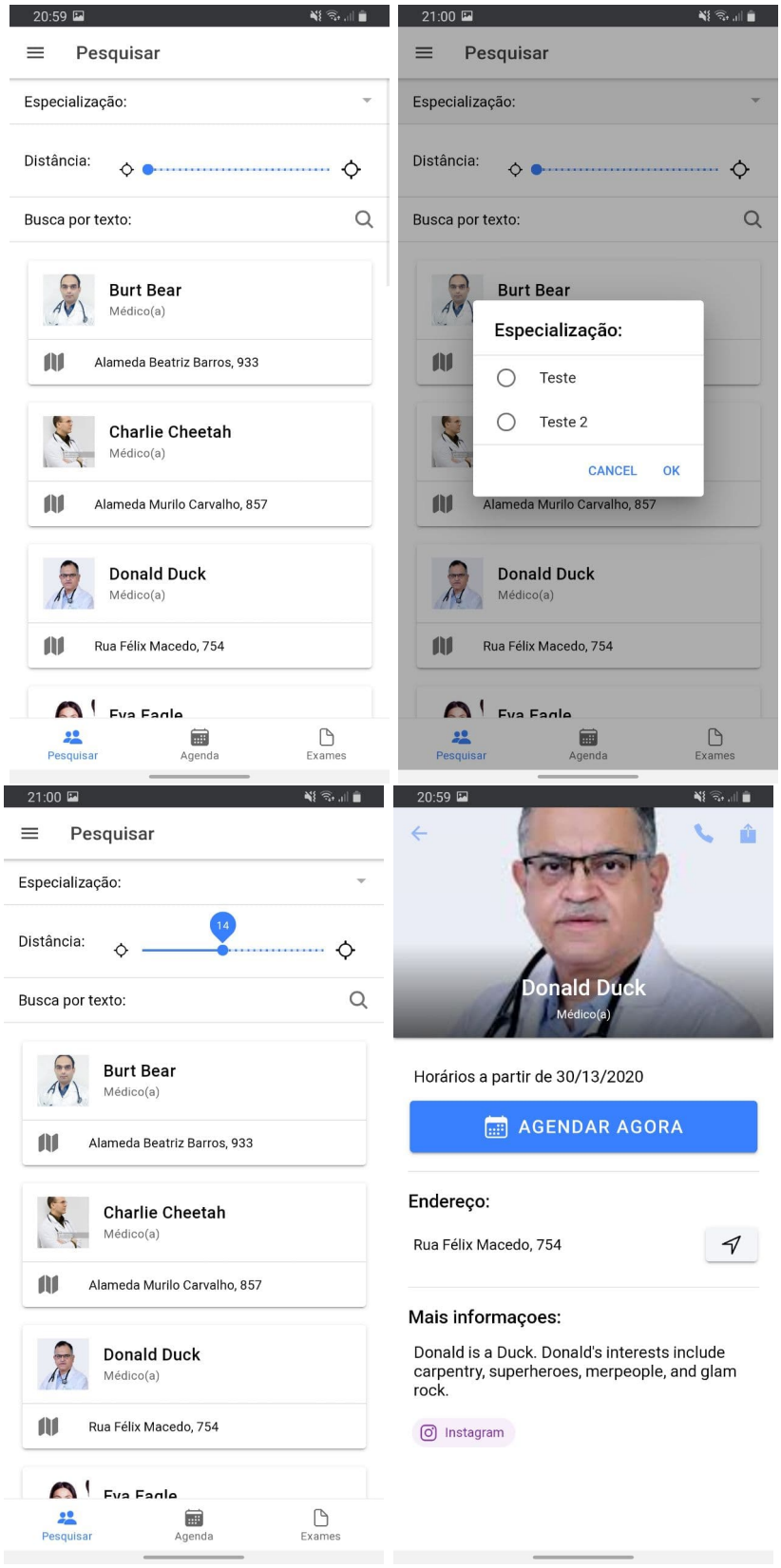
Como o provedor de saúde possui outras intenções de uso do sistema, o protótipo reflete essa outra experiência. Na Figura 34, vê-se as opções de uso para um provedor, seja mostrando as consultas, dando possibilidade de cancelá-las, dar *feedback* a elas, ver informações e exames do pacientes e marcando manualmente compromissos com usuários que não utilizam o sistema.

## 5.4 Arquitetura

A primeira decisão da arquitetura do sistema para o projeto é uma simples: em qual linguagem escrever a aplicação? Existem vantagens e desvantagens relacionado ao tipo de linguagem escolhido, e linguagens de tipagem estática e dinâmica parecem intercalar sua dominância (MARTIN, 2016b). Após uma análise, foi escolhido que se trabalharia com linguagem de tipagem estática para obter a vantagem de se saber da compatibilidade de tipos ainda em tempo de compilação.

Foram consideradas as linguagens mais populares em 2020, elencadas por (JOHNSON, 2020). *JavaScript* e *Python* foram descartados. Esse descarte se decorreu pela falta

Figura 30 – Busca de provedores do Protótipo de Alta Fidelidade




As telas de resultado de pesquisa por provedor e aplicações de filtros

Figura 31 – Marcação e cancelamento de consultas do paciente no protótipo de alta fidelidade

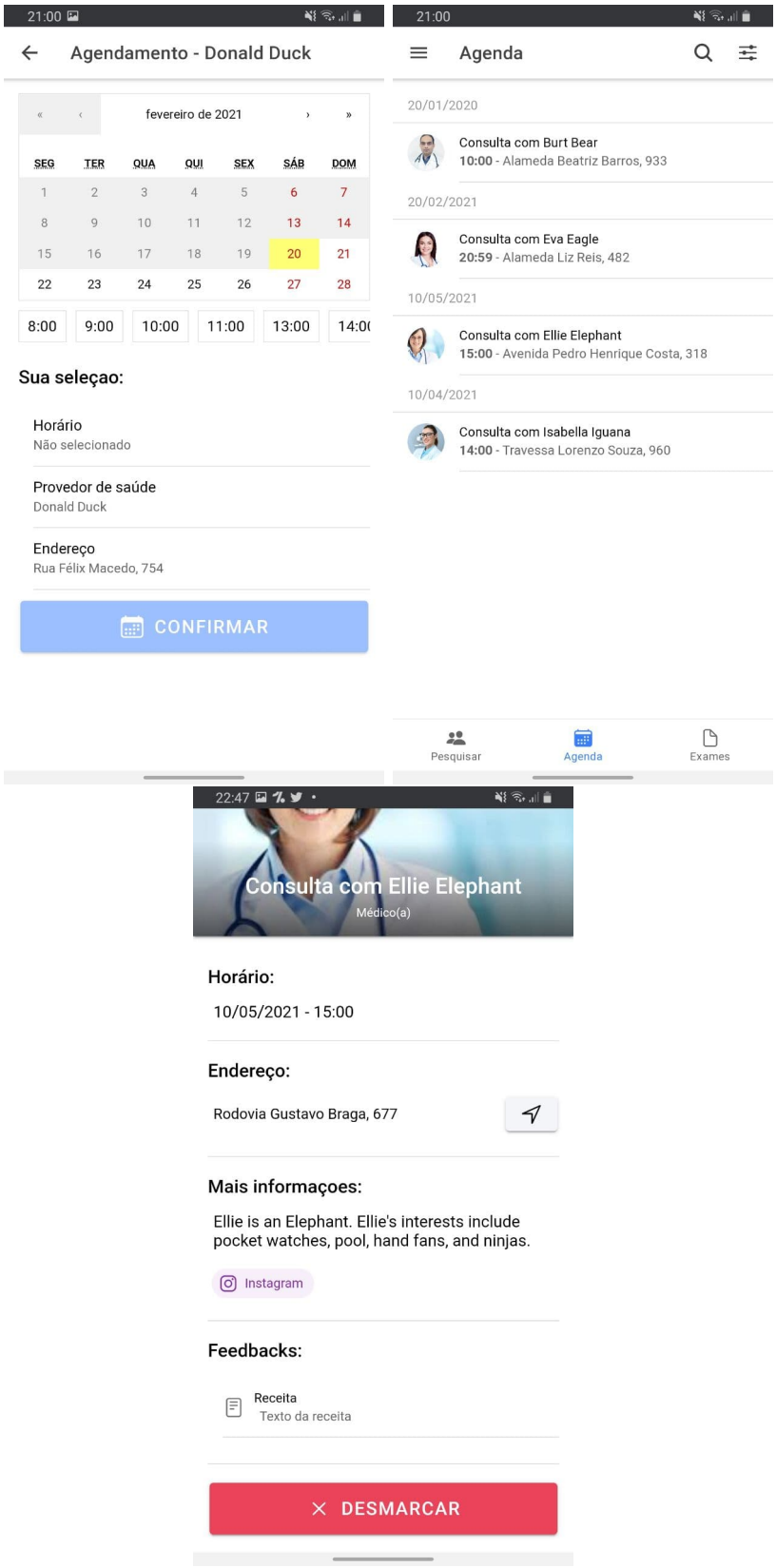
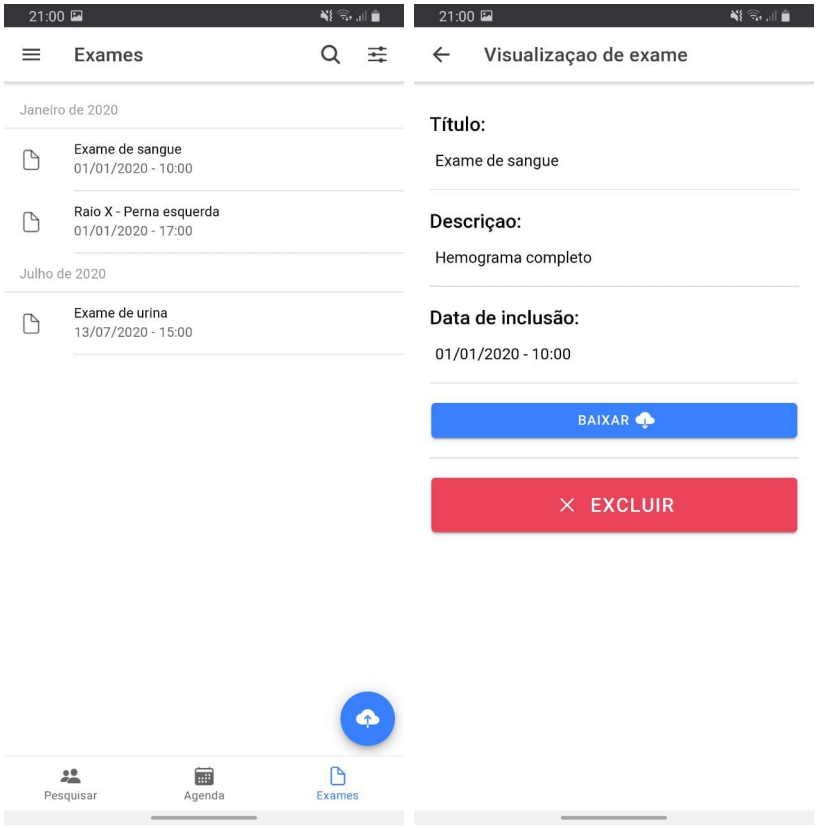


Figura 32 – Registro, remoção e visualização de exames no protótipo de alta fidelidade




de familiaridade do desenvolvedor com a linguagem, uma vez que ambas não dão um suporte nativo a interfaces que seja tão intuitivo como do Java.

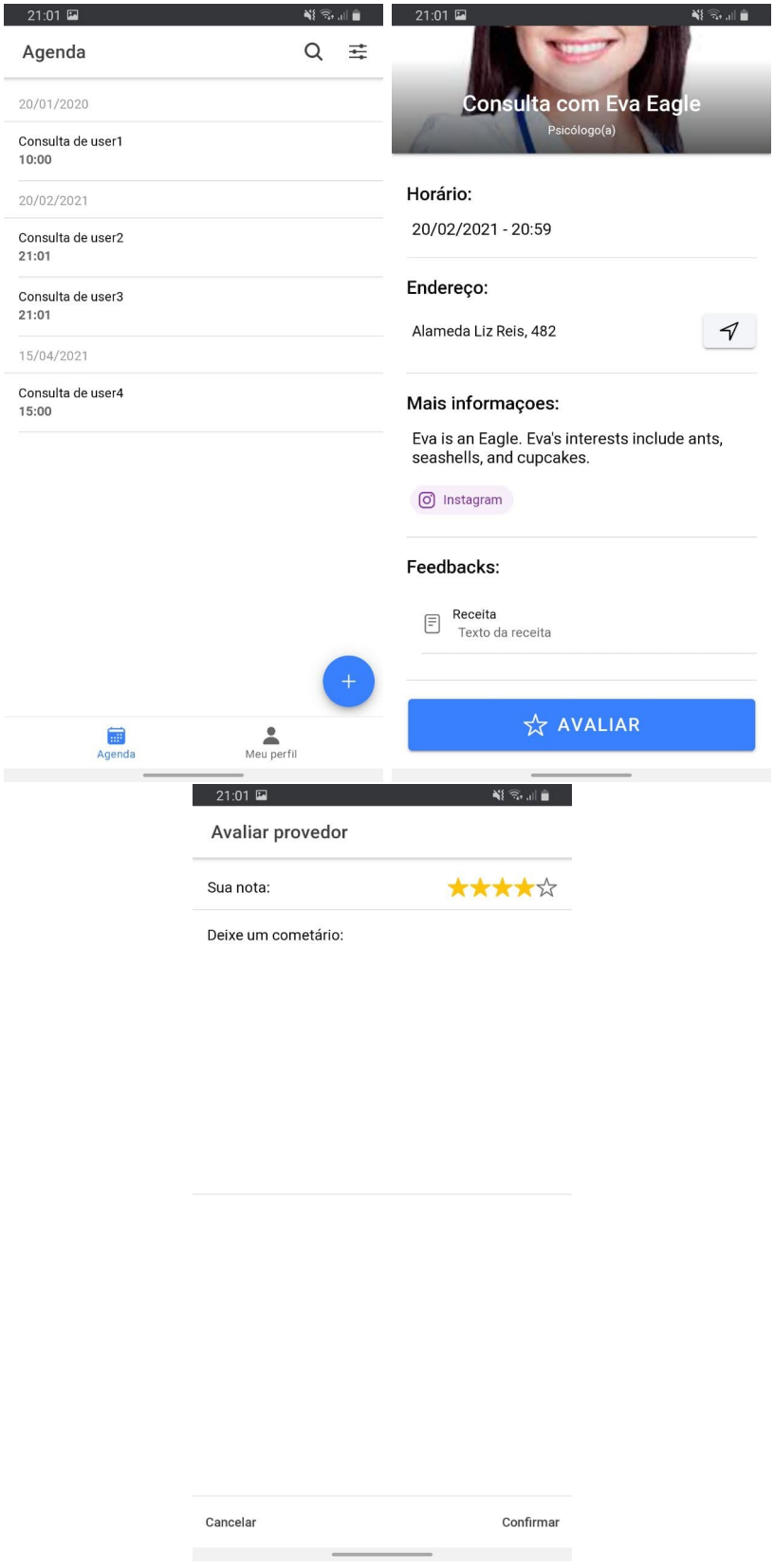
A competição entre as linguagens ficou entre Java e *TypeScript*. A princípio, havia sido escolhido a última por ser mais nova e estar com popularidade crescente (JOHNSON, 2020). Porém, como se desejava desenvolver em TDD, seria necessário configurar o uso do *framework* de testes e da linguagem.

*TypeScript* normalmente não é executado como linguagem final, mas como uma linguagem compilada em JavaScript, que é por sua vez utilizada como linguagem, a ser interpretada pelo NodeJS ou algum navegador (FERNANDES, 2019). Para se utilizar TypeScript, seria necessário configurar um *pipeline* de modo a compilar tanto o código, quanto os testes para executá-los, embrulhando os resultados para melhor entendê-los, por exemplo, em uma IDE.

Durante o desenvolvimento, gastou-se mais tempo do que o desejado fazendo as configurações necessárias para esse *pipeline*. Para não prejudicar o andamento do projeto, optou-se por trocar a linguagem do projeto para Java, uma vez que é uma linguagem bem consolidada no mercado, e sua tecnologia de testes, o jUnit, já vem integrado nas principais IDEs como Eclipse e IntelliJ IDEA (MEDEIROS, 2012).

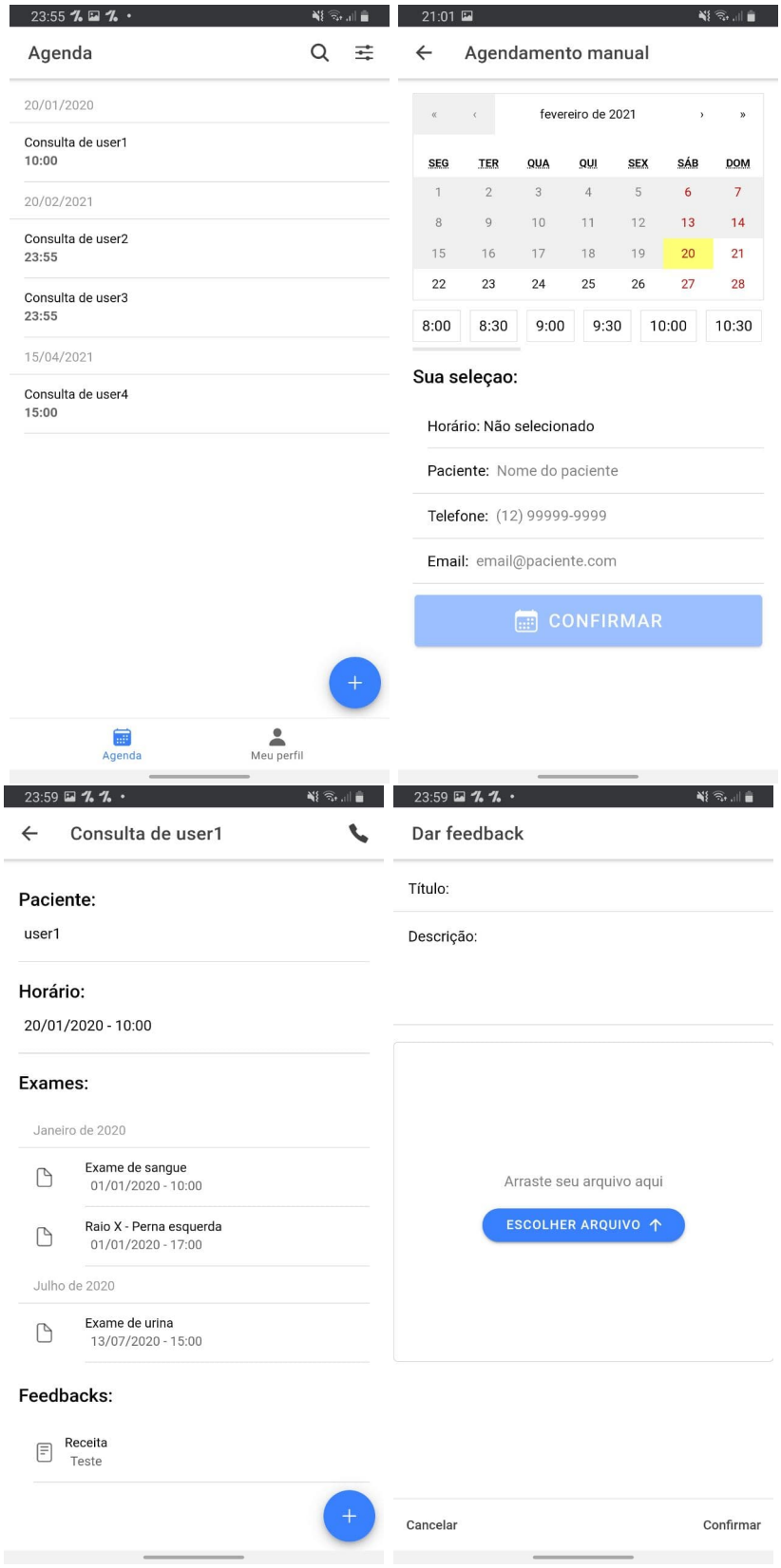
O código do projeto se encontra em <https://github.com/GustavoOS/med-plus>.

Figura 33 – Revisão de consultas do paciente no protótipo de alta fidelidade




Visualização de consultas passadas, com suas recomendações e possibilidade de avaliação de experiência com provedor de saúde.

Figura 34 – Opções de atendimento para o provedor no protótipo de alta fidelidade




Visualização e adição de compromissos e *feedback*

| Pacote | Ca | Ce | Na | Nc |
|---|---|---|---|---|
| Interface | 0 | 2 | 0 | 2 |
| Casos de Uso | 3 | 3 | 3 | 4 |
| Gateway | 0 | 1 | 0 | 1 |
| Exam Adder | 2 | 2 | 2 | 3 |
| Domínio | 6 | 0 | 4 | 4 |
| Patient Imp | 0 | 1 | 0 | 1 |
| Exam Impl | 0 | 2 | 0 | 2 |
| Appointment Impl | 0 | 1 | 0 | 1 |

Tabela 2 – Métricas primárias de Ca, Ce, Na e Nc dos pacotes do subsistema de adição de exames.

Está hospedado no Github, plataforma que além de hospedar código ainda utiliza o Git, famoso software de versionamento de código.

Devido à aplicação da filosofia TDD, houve um grande número de testes. Ao todo, foram feitos 172 testes, e a cobertura do código é de 100%. Os testes de aceitação foram automatizados, fazendo parte da estrutura de teste.

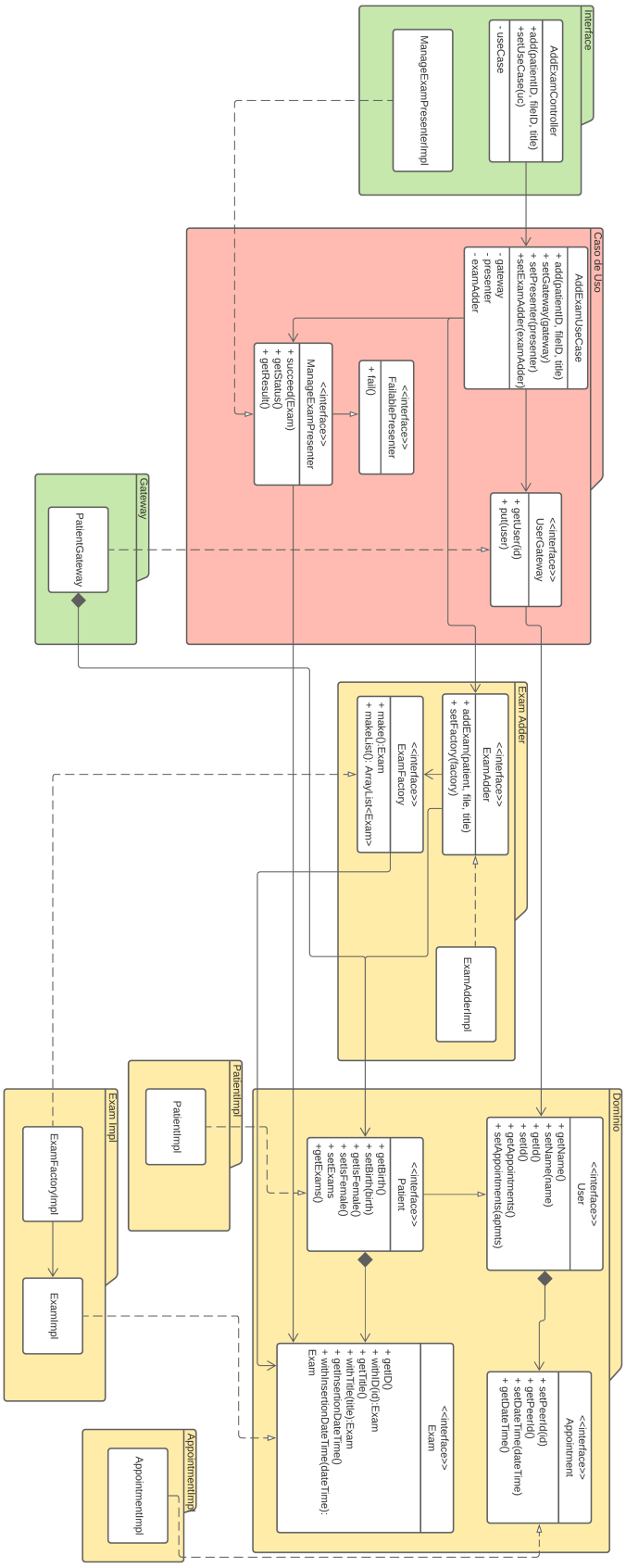
Na Figura 35, vemos um trecho de um diagrama de classes do sistema sobreposto com um diagrama de pacotes, da parte responsável da funcionalidade de adição de exames, desde a recepção do sistema de entrega, passando pelas entidades e *gateways*. Nesse diagrama, não está explícita a divisão de camadas e pacotes.

A partir da funcionalidade mapeada pela Figura 35 foi possível retirar métricas para a arquitetura de distribuição de classes em pacotes. As métricas primárias estão dispostas na 2 e foram obtidas a partir da contagem direta das classes e suas dependências dispostas nos pacotes mostrados na figura em questão.

As métricas primárias foram utilizadas como base de cálculo para métricas secundárias de abstração, instabilidade, distância para a sequência principal e distância normalizada para a mesma sequência, e estão dispostos na Tabela 3. Estas métricas foram utilizadas para a construção de um diagrama de dispersão da abstração pela instabilidade do subsistema, representado pela Figura 36.

A partir das métricas, pode-se afirmar que as principais proposição da arquitetura limpa estão sendo obedecidas: dependências estão seguindo em direção à estabilidade, e a grande maioria dos pacotes estão na extremidades da sequência principal. Dois pacotes, o de Casos de Uso e o *Exam Adder* estão próximos à sequência principal, mas fora dela. Tais pacotes estão mais próximos da Zona de Inutilidade do que da Zona de Sofrimento.

Figura 35 – Diagrama de Pacotes para funcionalidade de adição de exames



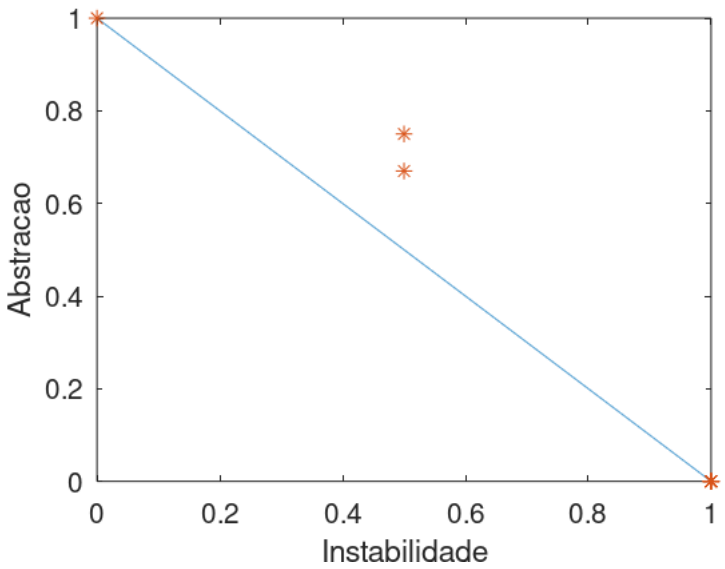

Nesse diagrama, existe uma sobreposição do diagrama de classes com o de pacotes.

| Pacote | Instabilidade | Abstração | Distância | Distância Normalizada |
|---|---|---|---|---|
| Interface | 1 | 0 | 0 | 0 |
| Casos de Uso | 0,5 | 0,75 | 0,18 | 0,25 |
| Gateway | 1 | 0 | 0 | 0 |
| Exam Adder | 0,5 | 0,67 | 0,12 | 0,17 |
| Domínio | 0 | 1 | 0 | 0 |
| Patient Imp | 1 | 0 | 0 | 0 |
| Exam Impl | 1 | 0 | 0 | 0 |
| Appointment Impl | 1 | 0 | 0 | 0 |

Tabela 3 – Métricas de abstração e instabilidade de componentes subsistema de adição de exames.

Métricas secundárias, obtidas a partir dos dados da Tabela 2.

Figura 36 – Diagrama de dispersão da abstração pela instabilidade do subsistema.




Em azul, a linha que define sequência principal. Em laranja, os valores apresentados pelos componentes.

## 5.5 Teste da Arquitetura

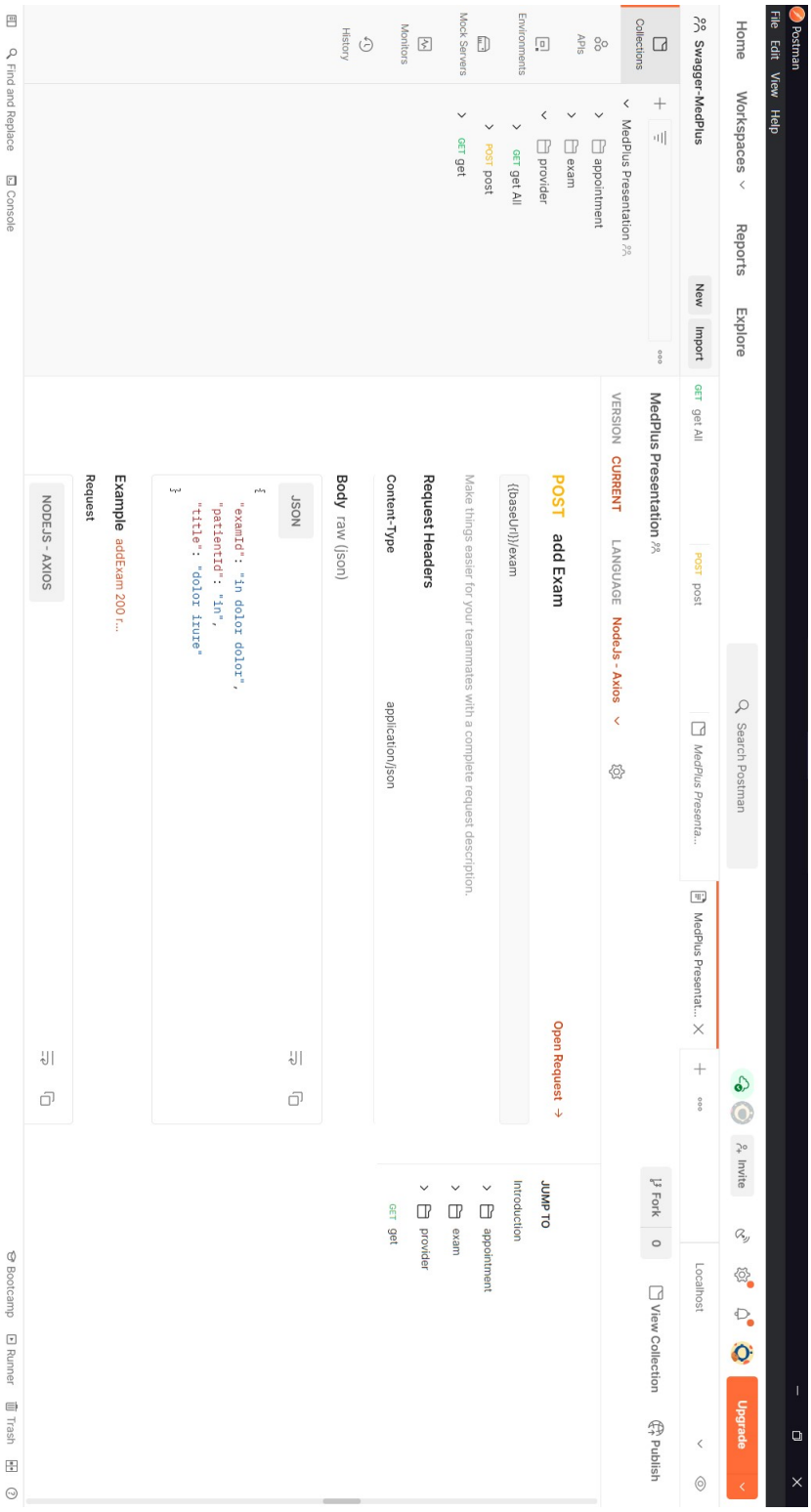
Um dos pontos principais da Arquitetura Limpa é que o desacoplamento possibilitaria que a aplicação pudesse funcionar de maneira independente da entrega. Para validar esse conceito, exportou-se toda a aplicação desenvolvido em um arquivo JAR. A partir dele, criou-se uma segunda aplicação, que utiliza as tecnologias abaixo para a construção de uma API REST, com implementação com banco de dados real:

- *Micronaut*
- *Hibernate*
- *MySQL*
- *Swagger*
- *Maven*

Esses *frameworks* produziram um subproduto interessante: uma especificação no formato OpenAPI 3. Essa especificação quando importada pelo *software Postman* gerou uma coleção de requisições com todos os campos e uma documentação capaz de informar qualquer potencial cliente da API. Essa documentação está parcialmente exibida pela Figura 37.

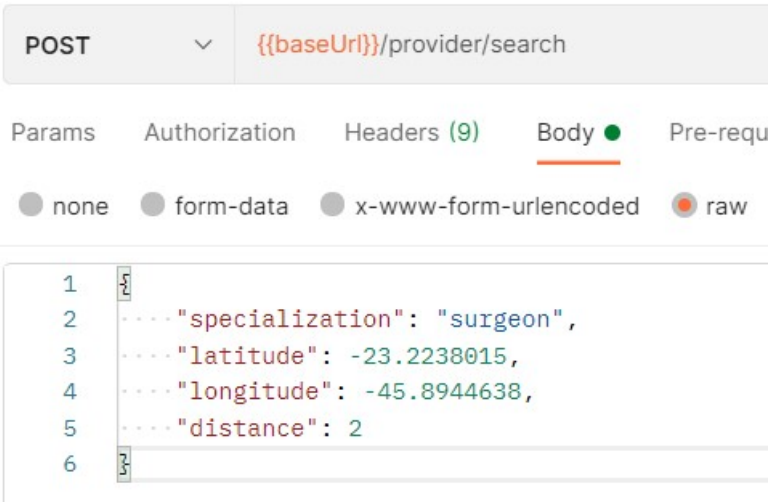
Foram realizados alguns testes manuais para verificar a funcionalidade. Os testes se baseavam em montar uma requisição, como na Figura 38, enviá-la para o servidor e verificar sua resposta, como na Figura 39. A parte final dos testes manuais é verificar se o que está sendo exibido está de acordo com os dados armazenados no banco de dados, como exibido na Figura 40.

Figura 37 – Trecho da documentação gerada pelo *Postman*




Documentação criada a partir da especificação gerada pelo *Swagger*

Figura 38 – Requisição feita ao servidor




Requisição feita ao servidor solicitando cirurgiões a um raio de 2km de uma localização.

Figura 39 – Resposta do servidor


```
[
    {
        "name": "Silva",
        "specialization": "surgeon",
        "id": "c43a58c6-de7b-4a95-8b41-b1b29e786422",
        "socialMediaURL": "silva.com"
    },
    {
        "name": "Joe",
        "specialization": "surgeon",
        "id": "da2ed3e9-566b-4521-8002-6e15f6f9958d",
        "appointments": [
            {
                "peerID": "4f24bdb4-4f0c-4d85-b8b4-44f757ba1bb1",
                "dateTime": "2021-03-01T14:47:53.559189"
            },
            {
                "peerID": "4f24bdb4-4f0c-4d85-b8b4-44f757ba1bb1",
                "dateTime": "2021-03-01T15:47:53.559189"
            },
            {
                "peerID": "0c9dbc30-2874-4983-a0b7-6885c409ddbc",
                "dateTime": "2021-03-02T10:47:53.559189"
            },
            {
                "peerID": "0c9dbc30-2874-4983-a0b7-6885c409ddbc",
                "dateTime": "2021-03-02T11:47:53.559189"
            },
            {
                "peerID": "4f24bdb4-4f0c-4d85-b8b4-44f757ba1bb1",
                "dateTime": "2021-03-02T14:47:53.559189"
            },
            {
                "peerID": "0c9dbc30-2874-4983-a0b7-6885c409ddbc",
                "dateTime": "2021-03-03T10:47:53.559189"
            }
        ],
        "socialMediaURL": "blabla.com"
    }
]
```


Resposta do servidor à requisição ilustrada na Figura 38

Figura 40 – Banco de Dados

| id | nom | latitude | longitude | mídia social | especializaçã |
|---|---|---|---|---|---|
| 7b11fdbb-0894-4e4b-afaf-880738c84f4c | Paz | -229546314 | -458329673 | paz.com | surgeon |
| 81a5cb24-d4ba-4789-b5da-3e76ae7ca551 | Benedito | -232232479 | -458934114 | sera.com | gynecologyst |
| c43a58c6-de7b-4a95-8b41-b1b29e786422 | Silva | -232173173 | -458918382 | silva.com | surgeon |
| da2ed3e9-566b-4521-8002-6e15f6f9958d | Joe | -232191582 | -458843743 | blabla.com | surgeon |

Dados armazenados na tabela do banco de dados que lista os provedores de saúde

# 6 Conclusão

Desenvolvimento de software que pode ser adotado pelos usuários é uma tarefa multidisciplinar, mesmo com aplicação de técnicas ágeis. Envolve pesquisa de mercado, envolvimento com o cliente e o consequente entendimento das intenções dele, negociação, gestão de projetos, gestão de qualidade, padrões de projeto, gestão de pessoas, entre outras.

Essa vivência multidisciplinar foi contemplada ricamente neste trabalho. Mesmo já tendo alguma bagagem profissional no ramo do desenvolvimento, dificilmente um desenvolvedor irá trabalhar tantos aspectos pois em empresas é natural que haja divisão de trabalho.

Um aspecto que o autor achou peculiar foi como os pontos de complexidade vindas da análise preliminar do programador se encaixaram sublimemente na gestão do projeto. Outra surpresa foi como os usuários, que recomendaram a existência da US que permite a um Paciente avaliar um Provedor de Saúde, classificaram em baixa prioridade essa *feature*.

A gestão de projetos, foi um pouco diferente da gestão feita por um GP convencional devido à alocação de um único recurso, o que trouxe alguns desafios metodológicos. Uma delas foi de definir, com um cronograma tão apertado, o momento de dividir a arquitetura em diferentes componentes. Enquanto as metodologias ágeis defendem que essa divisão deveria ocorrer de forma incremental, em cada iteração, foi-se decidido seguir o exemplo dado em (MARTIN; MARTIN, 2011), onde primeiro o sistema foi modelado, para então ser dividido em pacotes.

Essa decisão se mostrou errada, pois uma grande dificuldade do projeto foi que mais USs iriam sendo agregadas ao projeto, mais difícil seria distribuir o código-fonte em componentes. Esse obstáculo poderia ser explicado como decorrente da própria aplicação de princípios como SRP e REP, embora essa afirmação careça de provas.

Ao se utilizar ferramentas de engenharia reversa para verificar a modelagem UML do sistema desenvolvido, o resultado foi um diagrama grande demais para se trabalhar com divisão de pacotes, ainda que as camadas estivessem previamente separadas. Por esse motivo que a demonstração da arquitetura foi feita com um segmento do sistema, representado pela Figura 35.

O uso do diagrama de pacotes se mostrou bastante útil para a classificação dos componentes, principalmente devido à possibilidade de cálculo da abstração e da instabilidade dos pacotes. Esses dados foram extremamente úteis, por exemplo, na determinação que a classe *ExamAdderImpl* pertenceria ao componente *Exam Adder* pois isso equilibraria a instabilidade com o nível de abstração, aproximando o componente da sequência principal.

Contudo, parece sensato dizer que a modelagem poderia ser melhorada, por exemplo, quanto ao pacote do domínio da aplicação.

O maior feito da aplicação da arquitetura limpa é como a aplicação parece se tornar uma espécie de *software plug-and-play*. O programador pode escolher se vai ser executada como um aplicativo de linha de comando, como um programa Java clássico, ou se vai utilizar uma estrutura moderna de *API*. Todos os *gateways* foram pensados que podem ser feitos tanto em banco de dados relacionais quanto não relacionais. Todas as escolhas relacionadas a entrega da funcionalidade podem ser substituídas.

Após o desenvolvimento da segunda aplicação, que implementa uma *API REST*, foi possível comprovar que esse desacoplamento é possível. O sistema funcionou se integrando à primeira aplicação, dando a ela um servidor e os testes sugerem que está funcional.

Em uma eventual continuação deste projeto, poderá ser feita codificação e integração de um sistema de entrega semelhante ao protótipo de alta fidelidade. O cronograma poderá ainda ser seguido para aquisição das demais funcionalidades. A análise feita para o caso de uso de adição para exames pode ser estendida aos demais para aproximar os demais componentes da sequência principal.

Um ponto levantado é que um software que analisa os pacotes do projeto e automaticamente já elencasse as classes concretas e abstratas de cada componente, e definisse as métricas de abstração e estabilidade para cada componente poderia ajudar bastante uma divisão de componentes mais efetiva. Essa funcionalidade poderia ajudar desenvolvedores a serem mais efetivos na separação de componentes.

# Referências


ALLIANCE, A. User story template. 2001. Disponível em: <https://www.agilealliance.org/glossary/user-story-template/>. Citado na página 23.

ALLIANCE, A. *TDD Glossary*. 2020. Disponível em: <https://www.agilealliance.org/glossary/tdd>. Citado na página 42.

ALVES, P. *Dia da Informática: confira a história do computador e sua evolução*. TechTudo, 2014. Disponível em: <http://www.techtudo.com.br/noticias/noticia/2014/08/dia-da-informatica-confira-historia-do-computador-e-sua-evolucao.html>. Citado na página 15.

ASQ. American Society for Quality, 2021. Disponível em: <https://asq.org/quality-resources/quality-management-system>. Citado na página 30.

BABICH, N. *Prototyping 101: The Difference between Low-Fidelity and High-Fidelity Prototypes and When to Use Each*. Adobe Blog, 2017. Disponível em: <https://blog.adobe.com/en/publish/2017/11/29/prototyping-difference-low-fidelity-high-fidelity-prototypes-use.html>. Citado na página 27.

BECK, C. A. K. *Extreme programming explained: embrace change*. 2. ed. [S.l.]: Addison-Wesley Professional, 2004. ISBN 0321278658,9780321278654. Citado 2 vezes nas páginas 17 e 28.

BECK, K. *Test-Driven Development By Example*. [S.l.]: Addison Wesley, 2002. ISBN 978-0321146533. Citado na página 41.

CARROLL, J. M. Making use: A design representation. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 37, n. 12, dez. 1994. ISSN 0001-0782. Citado na página 23.

COCKBURN, A. *Writing Effective Use Cases*. 1. ed. [S.l.]: Addison-Wesley Professional, 2000. ISBN 0201702258,9780201702255. Citado 3 vezes nas páginas 20, 23 e 24.

COCKBURN, A. Hexagonal architecture. 2005. Disponível em: <https://web.archive.org/web/20180822100852/http://alistair.cockburn.us/Hexagonal+architecture>. Citado na página 39.

COHN, M. *User Stories Applied: For Agile Software Development*. [S.l.]: Addison-Wesley Professional, 2004. (Addison-Wesley Signature Series). ISBN 0-321-20568-5. Citado 16 vezes nas páginas 15, 17, 18, 20, 21, 22, 23, 25, 26, 27, 28, 29, 45, 48, 49 e 51.

COMMUNITY, C. W. E. *CWE-598: Use of GET Request Method With Sensitive Query Strings*. The MITRE Corporation, 2020. Disponível em: <https://cwe.mitre.org/data/definitions/598.html>. Citado na página 44.

CONSORTIUM, D.; STAPLETON, J. *DSDM: Business Focused Development*. 2. ed. [S.l.]: Pearson Education, 2003. ISBN 0321112245,9780321112248. Citado na página 28.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, C. S. R. L. *Introduction to algorithms*. 2. ed. [S.l.]: The MIT Press, 2002. ISBN 0262032937,9780262032933,0262531968,9780262531962,0070131511,9780070131514. Citado na página 40.

COUNCIL, M. S. *Trawling is common worldwide due to its efficiency in capturing large numbers of fish*. 2020. Disponível em: <https://www.msc.org/what-we-are-doing/our-approach/fishing-methods-and-gear-types/demersal-or-bottom-trawls>. Citado na página 25.

DAVIDSE, J. *API fundamentals: Understand both the basics and benefits of API development*. IBM Developer, 2020. Disponível em: <https://developer.ibm.com/articles/api-fundamentals/>. Citado 2 vezes nas páginas 42 e 43.

DEVMEDIA. Quem quer ser um programador fullstack? 2020. Disponível em: <https://www.devmedia.com.br/quem-quer-ser-um-programador-fullstack/38786>. Citado na página 22.

ELLIOTT, E. *Testing Software: What is TDD?* Medium, 2020. Disponível em: <https://medium.com/javascript-scene/testing-software-what-is-tdd-459b2145405c>. Citado na página 41.

ENDEAVOR. Endeavor Brasil, 2020. Disponível em: <https://endeavor.org.br/estrategia-e-gestao/pdca/>. Citado na página 30.

FARAH, A. *Como surgiu a Engenharia?* Instituto de Engenharia, 2019. Disponível em: <https://www.institutodeengenharia.org.br/site/2019/03/29/como-surgiu-a-engenharia/>. Citado na página 15.

FERNANDES, D. *TypeScript: Vantagens, mitos, dicas e conceitos fundamentais*. Rocketseat, 2019. Disponível em: <https://blog.rocketseat.com.br/typescript-vantagens-mitos-conceitos/>. Citado na página 65.

FIELDING, R. T.; TAYLOR, R. N. *Architectural Styles and the Design of Network-Based Software Architectures*. University of California, Irvine, 2000. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Citado na página 43.

FOWLER, M. *Modelo de maturidade de Richardson – os passos para a glória do REST*. 2010. Disponível em: <https://boaglio.com/index.php/2016/11/03/modelo-de-maturidade-de-richardson-os-passos-para-a-gloria-do-rest/>. Citado na página 44.

FOWLER, M. Software architecture guide. 2019. Disponível em: <https://martinfowler.com/architecture/>. Citado na página 39.

FUCCI, D. et al. *An External Replication on the Effects of Test-driven Development Using a Multi-site Blind Analysis Approach*. 2016. Disponível em: <http://people.brunel.ac.uk/~csstmms/FucciEtAl_ESEM2016.pdf>. Citado na página 42.

GLAS, M.; ZIEMER, S. Challenges for agile development of large systems in the aviation industry. Association for Computing Machinery, New York, NY, USA, 2009. Disponível em: <https://doi.org/10.1145/1639950.1640054>. Citado na página 17.

GONÇALVES, V. *Kaizen: como surgiu, etapas e como aplicar*. Blog Voitto, 2017. Disponível em: <https://www.voitto.com.br/blog/artigo/o-que-e-kaizen>. Citado na página 30.

GUDGIN, M. et al. *SOAP Version 1.2 Part 1: Messaging Framework*. W3C, 2007. (W3C Recommendation). Disponível em: <https://www.w3.org/TR/soap12/>. Citado na página 43.

HAT, I. R. *O que é API?* Red Hat, Inc, 2021. Disponível em: <https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces>. Citado na página 42.

HUNT, A.; THOMAS, D. *The Pragmatic Programmer: From Journeyman to Master*. [S.l.]: addison-wesley, 1999. ISBN 9780201616224,020161622X. Citado na página 53.

IBRAGIMOVA, E. *High-fidelity prototyping: What, When, Why and How?* Prototypr, 2016. Disponível em: <https://blog.prototypr.io/high-fidelity-prototyping-what-when-why-and-how-f5bbde6a7fd4>. Citado na página 61.

IEEE. IEEE guide for software requirements specifications. *IEEE Std 830*, Institute of Electrical and Electronics Engineers, 1998. Disponível em: <https://standards.ieee.org/standard/830-1998.html>. Citado na página 20.

INSIDER, C. *Vire expert na metodologia PDCA e melhore seus resultados*. Insider Store, 2020. Disponível em: <https://bit.ly/2OIZoll>. Citado na página 31.

INSTITUTE, P. M. *Um Guia do Conhecimento em Gerenciamento de Projetos (Guia PMBOK)*. 6. ed. [S.l.]: Project Management Institute, 2017. (Guia PMBOK). ISBN 1628251921,9781628251920. Citado 5 vezes nas páginas 15, 27, 28, 29 e 30.

INTEL. *Mais de 50 anos da Lei de Moore*. 2020. Disponível em: <https://www.intel.com/content/www/br/pt/silicon-innovations/moores-law-technology.html>. Citado na página 15.

JACOBSON, I. *Object-oriented software engineering: a use case driven approach*. Revised. [S.l.]: ACM Press; Addison-Wesley Pub, 1992. ISBN 9780201544350,0-201-54435-0. Citado 2 vezes nas páginas 23 e 39.

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. *The Unified Software Development Process*. Massachusetts, USA: Addison-Wesley, 1999. Citado na página 25.

JASPAN, C.; SADOWSKI, C. *No Single Metric Captures Productivity*. Apres, 2019. Disponível em: <https://doi.org/10.1007/978-1-4842-4221-6_2>. Citado na página 47.

JOHNSON, K. *GitHub: Python and TypeScript gain popularity among programming languages*. VentureBeat, 2020. Disponível em: <https://venturebeat.com/2020/12/02/github-python-and-typescript-gain-popularity-among-programming-languages/>. Citado 2 vezes nas páginas 62 e 65.

KOVITZ, B. L. *Practical Software Requirements: A Manual of Content and Style*. [S.l.]: Manning Publications, 1998. ISBN 1884777597,9781884777592. Citado na página 25.

LAUESEN, S. *Software Requirements: Styles & Techniques*. [S.l.]: Addison-Wesley Professional, 2002. ISBN 9780201745702,0201745704. Citado na página 22.

LEMONE, K. A. *Fundamentals of Compilers: An Introduction to Computer Language Translation*. USA: CRC Press, Inc., 1992. ISBN 0849373417. Citado na página 15.

LIMA, G. Diagrama de gantt: o que é e como utilizar? 2018. Disponível em: <https://www.voitto.com.br/blog/artigo/diagrama-de-gantt>. Citado na página 29.

LISKOV, B. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, v. 23, n. 5, p. 17–34, 3 1987. Citado na página 33.

MANIFESTO, T. A. *Princípios por trás do Manifesto Ágil*. 2001. Disponível em: <https://agilemanifesto.org/iso/ptbr/principles.html>. Citado na página 26.

MARTIN, R. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. 1. ed. [S.l.]: Pearson Education, 2017. (Robert C. Martin Series). ISBN 0134494326, 9780134494326. Citado 9 vezes nas páginas 15, 19, 32, 34, 36, 39, 41, 52 e 58.

MARTIN, R. C. *TDD Doesn't Work*. Clean Coder Blog, 2016. Disponível em: <https://blog.cleancoder.com/uncle-bob/2016/11/10/TDD-Doesnt-work.html>. Citado na página 42.

MARTIN, R. C. *Type Wars*. Clean Coder Blog, 2016. Disponível em: <https://blog.cleancoder.com/uncle-bob/2016/05/01/TypeWars.html>. Citado na página 62.

MARTIN, R. C. *TDD Harms Architecture*. Clean Coder Blog, 2017. Disponível em: <https://blog.cleancoder.com/uncle-bob/2017/03/03/TDD-Harms-Architecture.html>. Citado na página 42.

MARTIN, R. C.; MARTIN, M. *Princípios, padrões e práticas ágeis em C#*. [S.l.]: Bookman, 2011. ISBN 978-85-7780-841-0. Citado 15 vezes nas páginas 15, 17, 20, 23, 28, 29, 32, 33, 34, 35, 36, 38, 42, 53 e 75.

MEDEIROS, H. *Testes de Integração com Java e Junit*. DevMedia, 2012. Disponível em: <https://www.devmedia.com.br/testes-de-integracao-com-java-e-junit/25662>. Citado na página 65.

MEDICINA, C. F. de. *Resolução CFM nº 1643/2002*. 2002. Disponível em: <https://sistemas.cfm.org.br/normas/visualizar/resolucoes/BR/2002/1643>. Citado na página 45.

MEDICINA, C. F. de. *Resolução CFM nº 2227/2018*. 2018. Disponível em: <https://sistemas.cfm.org.br/normas/visualizar/resolucoes/BR/2018/222>. Citado na página 45.

MEDICINA, C. F. de. *Resolução CFM nº 2228/2019*. 2019. Disponível em: <https://sistemas.cfm.org.br/normas/visualizar/resolucoes/BR/2019/2228>. Citado na página 45.

METZ, C. Google is 2 billion lines of code—and it's all in one place. Wired, 2015. Disponível em: <https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/>. Citado 2 vezes nas páginas 15 e 32.

MOTROC, G. *The State of API Integration: SOAP vs. REST, public APIs and more*. 2017. Disponível em: <https://jaxenter.com/state-of-api-integration-report-136342.html>. Citado na página 43.

NELSON, B. J. *Remote procedure call*. 1981. Disponível em: <https://ia801900.us.archive.org/22/items/bitsavers_xeroxparctoteProcedureCall_14151614/CSL-81-9_Remote_Procedure_Call.pdf>. Citado na página 43.

NEWKIRK, J.; MARTIN, R. C. *Extreme Programming in Practice*. [S.l.]: Addison-Wesley, 2001. ISBN 0201709376, 978-0201709377. Citado na página 22.

NEWMAN, M.; LANDAY, J. Sitemaps, storyboards, and specifications: A sketch of web site design practice. p. 263–274, 01 2000. Citado na página 27.

NORTH, D. *Introduzindo o BDD*. Revista Better Software, 2006. Disponível em: <http://broncodev.com/2016-10-11-introduzindo-o-bdd/>. Citado na página 42.

PEARCE, J. The entity-control-boundary pattern. San Jose State University, 2015. Disponível em: <https://www.cs.sjsu.edu/~pearce/modules/patterns/enterprise/ecb/ecb.htm>. Citado na página 40.

REENSKAUG, T.; COPLIEN, J. O. The DCI architecture: A new vision of object-oriented programming. 2009. Disponível em: <https://www.artima.com/articles/dci_vision.html>. Citado na página 39.

REEVES, J. W. What is software design. *C++ Journal*, v. 2, n. 2, p. 14–12, 1992. Disponível em: <https://user.it.uu.se/~carle/softcraft/notes/Reeve_SourceCodeIsTheDesign.pdf>. Citado 2 vezes nas páginas 15 e 31.

ROBERTSON, J.; ROBERTSON, S. *Mastering the requirements process*. 1. ed. [S.l.]: Addison-Wesley, 1999. Citado na página 25.

ROBERTSON, J.; ROBERTSON, S. *Mastering the requirements process*. 3. ed. [S.l.]: Addison-Wesley, 2014. ISBN 978-0-321-81574-3,0-321-81574-2. Citado 2 vezes nas páginas 16 e 20.

RODRIGUEZ, A. *RESTful Web services: The basics*. IBM Developer, 2008. Disponível em: <https://cs.calvin.edu/courses/cs/262/kvlinden/references/rodriguez-restfulWS.pdf>. Citado na página 43.

SUTHERLAND, J. *Scrum: a arte de fazer o dobro do trabalho na metade do tempo*. 1. ed. [S.l.]: Casa da Palavra, 2016. ISBN 9788544104514. Citado na página 17.

WAKE, B. Invest in good stories and smart tasks. 2003. Disponível em: <https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/>. Citado 3 vezes nas páginas 21, 22 e 28.

WALKER, M.; TAKAYAMA, L.; LANDAY, J. High-fidelity or low-fidelity, paper or computer choosing attributes when testing web prototypes. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, v. 46, 09 2002. Citado 2 vezes nas páginas 26 e 27.

WINER, D. *XML-RPC for Newbies*. 1998. Disponível em: <http://scripting.com/davenet/1998/07/14/xmlRpcForNewbies.html>. Citado na página 43.

ZAK, P. *Measurement Myopia*. Drucker Institute, 2013. Disponível em: <https://www.drucker.institute/thedx/measurement-myopia/>. Citado na página 47.

# Anexos

# ANEXO A – Código

Este trabalho teve como produto o desenvolvimento de duas soluções de *software*. A primeira é uma solução da aplicação, que tange principalmente as regras de negócio. Seu código está hospedado em <https://github.com/GustavoOS/med-plus>.

A segunda solução de *software* é um sistema de entrega para a primeira aplicação, que utilizando as tecnologias Micronaut, Hibernate, MySQL, Swagger e Maven cria uma API REST para a aplicação relacionada. O código do segundo programa está hospedado em <https://github.com/GustavoOS/medplus-delivery>.